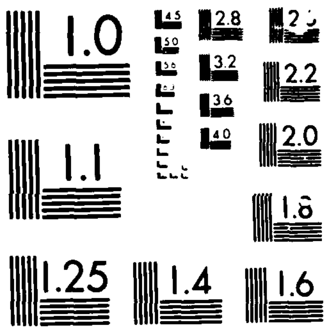
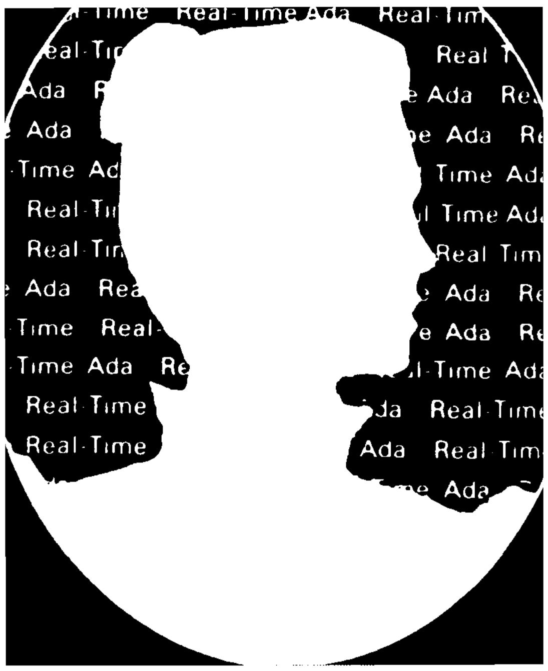
AD-A169 687 DESIGNING REAL-TIME SYSTEMS IN ADA (TRADEMARK)(U) 1/3
SOFTECH INC WALTHAM MA 1986

UNCLASSIFIED F/G 9/2 NL


1.0
4.5
2.8
5.0
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6


Real-Time Ada

(FINAL REPORT)

Real-Time Systems in Ada*

Abstract

Real-time software differs from other kinds of software in the sense that it must interact with external events. It must detect the occurrence of certain events as soon as they happen, and exercise control over external processes in a timely fashion. Real-time software must be cheap to produce and must be extremely reliable, even more so than other kinds of software. None of the existing approaches for real-time software design have been able to satisfy all of these requirements. ~~In~~ this report ~~we~~ evaluates, the role of Ada for this purpose and find that it too falls short. However Ada, unlike other approaches can make contributions towards reducing the cost and increasing the reliablity of real-time software. This report examines ideas and methods to be used in conjunction with Ada to satisfy the rest of the real-time requirements.

*Ada is a registered trademark of the US Government (Ada Joint Program Office)

CONTENTS

[✓] ☐ ☐

ltr. on file

Availability Codes	
Dist	Avail and / or Special
A-1	

CHAPTER 1

INTRODUCTION

1.1 REAL-TIME SYSTEM DESIGN CHALLENGE

A real-time software system is a system for monitoring and controlling a system of time dependent physical processes. The evolution of physical processes themselves are governed by their own deterministic physical laws. The real-time software system is designed to exercise some constraint on the direction of evolution of these processes. Thus some of the essential parameters of the physical processes are not controllable. Of this class of parameters the most significant one is the notion of time. In other words, the real-time system must sense the evolving parameters of the physical processes as soon as they are meaningful, and provide stimulus to the processes in a timely fashion. This sometimes imposes severe _timing requirements_ on the internals of the real-time system. The timing requirements can broadly be classified under three major categories: _deadline_, _throughput_, and _periodic processing_ requirements.

- _Deadline_ requirements dictate that the system be in a given state before some real time timing event occurs. This requires some form of synchronization with the given event.

- _Throughput_ requirements dictate that the overall system make progress at some given _rate_. This rate is usually measured and specified in various different ways: number of outputs generated per unit time, the ratio of number of outputs to the number of inputs if the number of inputs determine a well defined measure of time.

- _Periodic_ requirements dictate that a part or whole of system processing be repeated either indefinitely or a certain number of times at some specified rate.

In addition to the timing requirements the following are also important.

[1] Reliability: Like any any other software, real-time software must be designed to be reliable, perhaps even more so, since any timing errors could potentially have disastrous effects. By reliability we mean security from runtime and compile time errors. The range of possible errors that can arise is increased by the real-time nature of the system.

[2] Hardware Requirements: Choice of possible hardware equipment is often limited by considerations like size, power, availability, hardening, memory, and often politics. This further imposes limitations on the software in terms of program size.

1.2 HOW PREVIOUS EFFORTS HAVE MET THIS CHALLENGE

Most of the previously developed methods of real-time systems design have focused on two, often orthogonal aspects of the challenge.

1.2.1 Timing Focus

Typically, past efforts have been based on the concept of the cyclic executives. Starting from an initial functional decomposition of the problem into a set of tasks or jobs, the problem is reduced to that of finding a suitable ordering of these jobs that will meet the various timing requirements. Once a job schedule is defined, two logical steps follow. First, jobs are implemented that will stay within their required duration. Second, a cyclic executive is programmed which ensures that the jobs stay as close to the schedule as possible. Most of the effort is expended in designing and breaking up various jobs such that they stay within their required duration. A cyclic executive, serves mainly to meet the timing requirements of the system, in particular the deadline and the periodic requirements. The cyclic executive by itself is relatively easy to program and does not change all that much. It is the programming of various jobs that creates complications. As the coding of the jobs proceeds, it soon becomes apparent that the initial schedule requires massive changes. In an attempt to disrupt the initial design as little as possible, the system structure evolves in an unpredictable and uncontrollable manner. The upshot is that the system is very brittle and is a maintenance nightmare.

1.2.2 Functional Focus

Under this class of solutions, designs are formulated in terms of system functions. A more flexible scheduling mechanism, such as those provided in simple concurrent programming languages, is assumed. Also the concept of real-time is introduced by using a very simple language that is close to the machine (e.g. FORTRAN, C, MODULA etc.). These languages have simple enough features that can be implemented efficiently. Further improvements in the performance are introduced by using a "good" processor, production quality compilers, and all available optimization methods (see chapter 7). The resulting system is highly modular, easy to maintain and design. Such techniques, however, rarely work reliably for systems with hard real time deadlines. It is not to say that such techniques are not useful. For systems with soft real time requirements (e.g. lab instrumentations) cyclic based methods are an overkill and introduce strange complications. For slightly more effort than usual these functional design methods can work for certain classes of systems.

1.3 ROLE OF ADA IN REAL-TIME SYSTEMS DESIGN.

One of the major design goals of Ada was to promote the production of reliable, robust, and cost-effective software for embedded systems. Ada embodies many of the major software engineering principles through language features like <u>strong</u> <u>typing</u>, <u>separate</u> <u>compilation</u> facilities, <u>packages</u> and <u>generics</u>, <u>multitasking</u> and <u>exception</u> <u>handling</u>. All this leads to cost-effective and reliable programming solutions to many embedded systems.

1.4 SCOPE OF THIS REPORT

The purpose of this report is to investigate techniques that can help develop better real-time software in Ada. First, the old approaches to real time software development are investigated. Their strengths and weaknesses are evaluated. Finally, specific recommendations are made.

1.5 STRUCTURE OF THE REPORT

This report is divided into three parts. The main body of the report is contained in the first two parts. The first part (chapters 2-5) of the report surveys the current major methodology for developing real-time systems. Chapter 2 surveys the concept of cyclic executives, outlining the basic model of cyclic processing and some of its applications. Chapter 3 addresses the treatment of timing errors that arise from design mistakes or unanticipated runtime events. Treating these timing errors is a significant part of the real time design process. The issues are discussed in terms of a general fault-tolerant design framework and applied to cyclic based and data driven or functional methods of design. Chapter 4 describes the third major area of concern, known as mode changes. Modes are defined as often mutually conflicting schedules that cannot be active simultaneously for a variety of reasons. In this chapter we try to show that modes are unavoidable for any of the traditional design methods in use. Chapter 5 provides a summary of evaluations for the existing approaches to real time system designs and discuss the problems that arise due to the design methods themselves.

The second part of the report (chapters 6-8) outlines a framework for proposing specific solutions in Ada and follows some of the more promising approaches. Chapter 6 starts with a general set of requirements that must be addressed by any real time design methodology. Next, some of the proposals that arise naturally from the ensuing discussion are examined critically. Chapter 7 looks at the multitasking and synchronization aspects of Ada in a critical manner. Timing costs of some aspects of Ada are examined. Finally, some potential transformations that might be used to improve timing performance are examined. Chapter 8 addresses the issues regarding verification and validation of real-time requirements and specifications of systems. A variety of promising timing analysis and modeling approaches are surveyed from the perspective of applicability to Ada.

The third part (chapters 9-10) includes two case studies undertaken as part of this project. The first case study, takes a simple real time problem and analyzes two Ada solutions. The second case study solves a simpler subproblem of the first problem, emphasizing Ada from a reusability perspective.

CHAPTER 2

CYCLIC EXECUTIVES

Over the years, real-time programmers have developed the cyclic executive to provide a fast, efficient method for scheduling their processing. The cyclic executive has not only met many real time requirements but also brought with it many problems, especially in the area of program maintenance. This chapter examines the _cyclic executive concept_ in detail. The first thing to realize about cyclic executives is that there is no real standard. The term _cyclic executive_ refers to a concept used by many different real time designers in many different real time designs, resulting in many different cyclic executives. For the purposes of this report we will define a simple model upon which which most of the existing cyclic executive designs are based. Using this model as a basis we describe how different kinds of real-time systems designs are formulated. We use Ada for illustrative purposes only. Such use of Ada does not imply that this is how a cyclic executive based system might be implemented.

2.1 BACKGROUND

The entire area of cyclic executive design is based upon the theory of _deterministic scheduling_ or _resource allocation_.* This area of computer science is concerned with finding an _optimal_ or a _near-optimal_ schedule for a number of tasks or jobs for which a relative order of execution is known. The resulting schedule is meant to define times at which a given task is to be started. The basic idea is that an optimal or a near-optimal schedule often has a better chance of meeting the timing requirements than a non-optimal schedule. Time, however, is not the only parameter that determines an optimal schedule. Additional and equally important constraints are: number of processors, task duration, task interruptability, processor idleness, periodicity, deadlines, and resource limitations. The resulting schedules are

*A comprehensive survey of deterministic processor scheduling is given in [16].

meant to optimize certain measures of performance which include:

[1] Completion Time: This represents the duration of time for which a given schedule lasts. If the completion time is minimal, then the system throughput can be maximized, where throughput is defined to be the number of activities performed per unit time.

[2] Number of Processors: Minimizing the number of processors can lead to cost savings as well as allocation of remaining processors to fault-tolerant and other background functions.

[3] Mean Flow Time: Flow time represents the time for which a given task remains in memory awaiting execution. Minimizing this measure can lead to improved throughput as well as effective memory utilization.

[4] Processor Idle Time: For systems with hard real time requirements, it is sometimes necessary to minimize the average idle time for a given processor. For systems with hard real-time deadline requirements, this strategy may not work. Some simple heuristics, usually application dependent, determine whether or not it is worth optimizing this measure.

For many cases finding an optimal schedule is nearly impossible. In such cases a number of heuristic techniques are applied.

The resulting schedules, when they exist, are represented by what are known as Gantt Charts. A Gantt chart is a finite time line on which different jobs are represented as non-overlapping intervals.

In the design of a real-time system, the first step consists of defining an acceptable schedule that will satisfy the real-time constraints. The next step in the approach is to ensure that the resulting schedule is obeyed. There are two related implemention strategies that ensure this. First, jobs are implemented that will stay within their required duration. Second, a cyclic executive is programmed which ensures that the jobs stay as close to the schedule as possible. Most of the effort is expended in designing and breaking up various jobs such that they stay within their required duration.

Often, there is no single acceptable schedule that can be found. There may be more than one schedule that will work under restricted but non-overlapping conditions. In such cases each schedule is called a mode of operation or simply a mode. When at runtime some condition is detected, the new mode is determined and the processing is required to conform to the newly selected mode.

The use of modes impose further requirements on the design of the system. The cyclic executive must function under different schedules. Also the transition between modes requires careful analysis. The current processing often cannot simply be abandoned; it must be completed to a point where consistency of various data is restored. Furthermore, timing errors in a given mode can, in principle, contaminate the operation of the next mode. We will discuss these issues in detail in subsequent chapters, but at present we describe in detail the issues that dictate the design of cyclic processing executives.

2.2 CYCLIC PROCESSING MODEL

Most real-time system designs involve three different classes of processing: event driven, periodic or time-critical, and background processing.

- Event driven processing involves handling externally generated events such as interrupts.

- Background processing is usually comprised of self checking algorithms, or algorithms that are lengthy and time non-critical. Background generally takes the form of a non-terminating loop.

- Periodic Processing in a system contains the most time critical processing and has the most potential for variation. Deterministic processor scheduling techniques are commonly used here.

In our cyclic processing model the scheduling decisions are represented as consisting of one major schedule for each mode of operation. The major schedule corresponds to the time line of the Gantt Chart. Each major schedule is further divided into a set of minor schedules. Minor schedules represent the various non-overlapping tasks on the time lines. The primary unit of time is the minor cycle; a secondary unit of time is the major cycle. A major cycle is constrained to be an integer multiple of the minor cycle. During each minor cycle a number of routines are called. This set of routines is referred to as a Frame. The actual routines called can vary from frame to frame. Each frame, however, has a fixed set of routines to call. If a frame fails to complete within its minor cycle, it is terminated. All periodic processing is scheduled to complete at least once in each major cycle.

The primitives of our cyclic processing model are illustrated in Figure 2.1. The primitive data objects of this model are Frame_Type and Frame_Set. Frame_Type objects represent the routines performed within the given frame, while Frame_Set objects represent major schedules. The explicit notion of time is absent from this model, but it is enforced by the clock driven cyclic executive .

The baseline cyclic executive is illustrated in terms of our cyclic executive primitives in Figure 2.2.

The first set of major variations comes with the consideration of what actions to take when a frame overruns its minor cycle. We address three techniques: termination, suspension, amd overrun continuation. In our baseline executive, the overrunning frame is terminated, to be restarted at its next scheduled call. This presupposes that each routine called from this executive both keeps a running record of what processing it has accomplished and has some form of built-in protection for critical data storage operations. While both of these requirements are generally realizable (the first is usually not difficult to build into programs, while the second can be accomplished by temporarily disabling interrupts), they do not always result in the best design technique.

The alternatives to termination include suspending the frame's execution and allowing it to continue at the next possible moment (Figure 2.3), or allowing the overrunning frame to complete before starting the next frame while logging a fault to indicate the overrun (Figure 2.4). Additionally, some combination of the three techniques for handling overruns can be used. Overruns might be allowed in the minor cycles of a major cycle but not into the next major cycle, or overruns might be allowed as long as there are no more than a certain number of consecutive overruns (Figure 2.5). Finally, there might be a set pattern as to which frames are allowed to overrun and which frames are not (Figure 2.6). For each combination the decision must be made whether or not to terminate or suspend frames that are not allowed to overrun.

Frame suspension still requires a protection mechanism for critical memory storage and several other potentially unpleasant effects. If a suspended frame is allowed to complete after several other frames have run, a change in the order of execution will have resulted. This means synchronization is jeopardized, causing potential harm to data passing and data integrity. (Only software that can stand non-deterministic scheduling can stand to be suspended). In a frame suspension executive the designer must also decide what to do if the frame is still suspended when it is time for it to start again. Frame suspension also tends to take time away from background processing to complete suspended processing. Some executives that use this approach include a small run-burst for the background routines somewhere within the major cycle to guarantee the background routines some processing time.

```
package Frame_Manager is
   type Frame_Type is private;     -- contains several routines which
                                   -- comprise a frame.
   type Frame_Set  is private;     -- contains several frames which
                                   -- comprise a major cycle.

   function  Current_Frame return Frame_Type;
   -- this returns the frame which was started or resumed most
   -- recently

   procedure Set_Current_Frame (f : Frame_Type);
   -- set the current frame indicator

   function  Next_Frame return Frame_Type;
   -- this returns a frame which is to be started next.

   procedure Start_Frame (f : Frame_Type);
   -- starts the execution of a frame

   procedure Stop_Frame  (f : Frame_Type);
   -- stops the execution of this frame

   procedure Suspend (f : Frame_Type);
   -- suspends the execution of the frame

   procedure Resume (f : Frame_Type);
   -- resumes the execution of this frame

   function  Suspended_Frame return Frame_Type;
   -- returns the suspended frame that has been suspended longest

   function  Is_Suspended return  Boolean;
   -- if any frames are suspended

   function  Is_Running (f : Frame_Type) return  Boolean;
   -- if the frame f is still running

private
   .
   .
   .
end Frame_Manager;
```

Figure 2.1. Cyclic Executive Primitives

```
task This_Event is                 -- One Event Driven Process
  pragma priority(10);
  entry Event1;
end This_Event;

task body This_Event is
begin
  --Event1 processing
end This_Event;

task That_Event is                 -- Another Event Driven Process
  pragma Priority(10);
  entry Event2;
end That_Event;

task body That_Event is
begin
  --Event2 processing
end That_Event;


task Baseline is                  -- Stop Overrunning Frames
  pragma Priority(5);
  entry Tick;                        -- One Tick per minor cycle
end Baseline;

task body Baseline is
   Major_Cycle : Frame_Manager.Frame_Set;
begin
   Major_Cycle := List_of_all_the_frames_in_the_Major_Cycle;
   loop -- forever
      accept Tick;        -- it is time to start the next frame
      if Frame_Manager.Is_Running(Frame_Manager.Current_Frame) then
        Frame_Manager.Stop_Frame(Frame_Manager.Current_Frame);
            -- the frame which was last started overran; therefore
            -- stop it
      end if;
      Frame_Manager.Set_Current_Frame(Frame_Manager.Next_Frame);
      Frame_Manager.Start_Frame(Frame_Manager.Current_Frame);
   end loop;
end Baseline;
```

Figure 2.2. Baseline Cyclic Executive (1 of 2)

```
task Background is
  pragma Priority(0);
end Background;

task body Background is
  begin
    loop  -- forever
      -- Background Processing
    end loop;
  end Background;
```

Figure 2.2. Baseline Cyclic Executive (2 of 2)

```
task Variationl is                  -- Suspend Overrunning Frames
                                    -- Resume frames when there is time
  entry Tick;                         -- One Tick per minor cycle
  entry Frame_Done;                   -- Called by each frame as
                                      -- the frame finishes
end Variationl;


task body Variationl is
   Major_Cycle : Frame_Manager.Frame_Set;
begin
   Major_Cycle :=  List_of_all_the_frames_in_the_Major_Cycle;
   loop -- forever
     select
       accept Tick;      -- it is time to start the next frame
       if Frame_Manager.Is_Running(Frame_Manager.Current_Frame) then
         Frame_Manager.Suspend(Frame_Manager.Current_Frame);
             -- the frame which was last started overran; therefore
             -- suspend it
       end if;
       Frame_Manager.Set_Current_Frame(Frame_Manager.Next_Frame);
       Frame_Manager.Start_Frame(Frame_Manager.Current_Frame);
     or
       accept Frame_Done;
       if Frame_Manager.Is_Suspended then
         Frame_Manager.Set_Current_Frame(Frame_Manager.Suspended_Frame);
         Frame_Manager.Resume(Frame_Manager.Current_Frame);
       end if;
     end select;
   end loop;
end Variationl;
```

Figure 2.3. Frame Suspension Cyclic Executive

```
task Variation2 is                  -- Allow Overrunning Frames
                                    -- to complete but log the overrun
  entry Tick;                         -- One Tick per Minor Cycle
  entry Frame_Done;                   -- Called by each frame
                                      -- as frame finishes
end Variation2;


task body Variation2 is
   Major_Cycle : Frame_Manager.Frame_Set;
   Overrun : Boolean := False;
begin
   Major_Cycle := -- list of all the frames in the Major Cycle
   loop -- forever
     select
       accept Tick;         -- it is time to start the next frame
       if Frame_Manager.Is_Running(Frame_Manager.Current_Frame) then
         Overrun := True;
            -- the frame which was last started overran; therefore
            -- log it
       else
         Overrun := False;
         Frame_Manager.Set_Current_Frame(Frame_Manager.Next_Frame);
         Frame_Manager.Start_Frame(Frame_Manager.Current_Frame);
       end if;
     or
       accept Frame_Done;
       if Overrun then
         Frame_Manager.Set_Current_Frame(Frame_Manager.Next_Frame);
         Frame_Manager.Start_Frame(Frame_Manager.Current_Frame);
       end if;
     end select;
   end loop;
end Variation2;
```

Figure 2.4. Overrun Logging Cyclic Executive

```
task Variation3 is                 -- Allow Overrunning Frames
                                   -- unless there have been too
                                   -- many overruns in a row
  entry Tick;                        -- One Tick per Minor Cycle
  entry Frame_Done;                  -- Called by each frame
                                     -- as frame finishes
end Variation3;


task body Variation3 is
   Major_Cycle : Frame_Manager.Frame_Set;
   Overrun : Boolean := False;
   Terminated : Boolean := False;
   Overrun_Count : Integer := 0;
   Maximum_Allowable_Consecutive_Overruns: Integer := 4;
begin
   Major_Cycle := List_of_all_the_frames_in_the_Major_Cycle;
   loop -- forever
      select
        accept Tick;        -- it is time to start the next frame
        if Frame_Manager.Is_Running(Frame_Manager.Current_Frame) then
          Overrun := True;
          Overrun_Count := Overrun_Count + 1;
          if Overrun_Count > Maximum_Allowable_Consecutive_Overruns then
            Frame_Manager.Stop_Frame(Frame_Manager.Current_Frame);
            Terminated := True;
            Frame_Manager.Set_Current_Frame(Frame_Manager.Next_Frame);
            Frame_Manager.Start_Frame(Frame_Manager.Current_Frame);
          else
            Terminated := False;
          end if;
        else
          Overrun := False;
          Overrun_Count := 0;
          Frame_Manager.Set_Current_Frame(Frame_Manager.Next_Frame);
          Frame_Manager.Start_Frame(Frame_Manager.Current_Frame);
        end if;
      or
        accept Frame_Done;
        if Overrun and (not Terminated) then
          Frame_Manager.Set_Current_Frame(Frame_Manager.Next_Frame);
          Frame_Manager.Start_Frame(Frame_Manager.Current_Frame);
        end if;
      end select;
   end loop;
end Variation3;
```

Figure 2.5. Limited Consecutive Overrun Executive

```
task Variation4 is                 -- Allow Overrunning Frames
                                   -- in some cases
  entry Tick;                        -- One Tick per Minor Cycle
  entry Frame_Done;                  -- Called be each frame
                                     -- as frame finishes
end Variation4;


task body Variation4 is
   Major_Cycle : Frame_Manager.Frame_Set;
   Overrun: Boolean := False;
   Terminated: Boolean := False;
begin
   Major_Cycle := List_of_all_the_frames_in_the_Major_Cycle;
   loop -- forever
      select
        accept Tick;         -- it is time to start the next frame
        if Frame_Manager.Is_Running(Frame_Manager.Current_Frame) then
          Overrun := True;
          if not(Overrun_OK(Frame_Manager.Current_Frame))
            Terminated := True;
            Frame_Manager.Stop_Frame(Frame_Manager.Current_Frame);
            Frame_Manager.Set_Current_Frame(Frame_Manager.Next_Frame);
            Frame_Manager.Start_Frame(Frame_Manager.Current_Frame);
          else
            Terminated := False;
          end if;
        else
          Overrun := False;
          Frame_Manager.Set_Current_Frame(Frame_Manager.Next_Frame);
          Frame_Manager.Start_Frame(Frame_Manager.Current_Frame);
        end if;
      or
        accept Frame_Done;
        if Overrun and (not Terminated) then
          Frame_Manager.Set_Current_Frame(Frame_Manager.Next_Frame);
          Frame_Manager.Start_Frame(Frame_Manager.Current_Frame);
        end if;
      end select;
   end loop;
end Variation4;
```

Figure 2.6. Privileged Frame Overrun Executive

Allowing unrestricted overrun to occur will not meet the needs of systems with "hard" periodic requirements unless there is no possibility of frame overrun. This does limit the applicability of this executive. Where this executive is practical, however, it avoids the problems inherent in the first two appro hes.

The approaches where overruns are allowed in limited situations are useful in a variety of applications where it can be determined from the application when timing is the most critical. The result is additional work for the system designer in customizing the executive to his application. The system designer still faces the critical question of what to do if overruns occur when they are not allowed.

Some applications carry time slicing farther than the minor cycle, where each routine called in a frame is allocated a period of time to complete. Routines which overrun are either terminated or suspended, with the same advantages and disadvantages discussed for terminating or suspending minor frames. The implementation for this type of executive is similar to the privileged frame overrun executive shown in Figure 2.6.

A fourth processing structure is often introduced into the executive with a priority between the cyclic and background structures. This is generally called foreground, and it consists of processing which does not have to execute in real time but has higher priority than background processing. If neither event driven nor periodic processing is running, foreground will run until it runs out of things to do, then background will run. Figure 2.7 shows the Baseline Executive modified to include foreground processing. The foreground task shown is triggered every time that a higher priority task completes and that foreground is not already active. Each time it is triggered, it executes all routines that have work to do until all opportunity for work is exhausted. Once the available work is exhausted, the foreground blocks itself and allows background to run until the next foreground trigger.

The final alternative we will discuss is a situation where two or more cyclic structures of any of the above types are competing for the same resources with some mechanism for assigning disputed resources. Possible mechanisms would be cyclic structure priority, individual routine priority, or some sort of alternate use scheme.

The cyclic executive types described above are summarized in Table 2.1.

```
task This_Event is                -- One Event Driven Process
  pragma priority(10);
  entry Event1;
end This_Event;

task body This_Event is
begin
  -- Event1 processing
  Foreground_Trigger.Time_Slot;
end This_Event;

task That_Event is                -- Another Event Driven Process
  pragma Priority(10);
  entry Event2;
end That_Event;

task body That_Event is
  begin
  -- Event2 processing
  Foreground_Trigger.Time_Slot;
end That_Event;

task Baseline is                 -- Stop Overrunning Frames
  pragma Priority(5);
  entry Tick;
  entry Frame_Done;
end Baseline;

task body Baseline is
   Major_Cycle : Frame_Manager.Frame_Set;
begin
   Major_Cycle := List_of_all_the_frames_in_the_Major_Cycle;
   loop -- forever
     select
       accept Tick;        -- it is time to start the next frame
       if Frame_Manager.Is_Running(Frame_Manager.Current_Frame) then
         Frame_Manager.Stop_Frame(Frame_Manager.Current_Frame);
             -- the frame which was last started overran; therefore
             -- stop it
       end if;
       Frame_Manager.Set_Current_Frame(Frame_Manager.Next_Frame);
       Frame_Manager.Start_Frame(Frame_Manager.Current_Frame);
     or
       accept Frame_Done;
       Foreground_Trigger.Time_Slot;
     end select;
   end loop;
end Baseline;
```

Figure 2.7. Baseline Cyclic Executive with Foreground Processing (1 of 3)

```
task Foreground_Trigger is
  entry Time_Slot;
  entry Start_Foreground;
end Foreground_Trigger;

task body Foreground_Trigger is
    Start_OK: Boolean := False;
  begin
    loop   -- forever
      select
        accept Time_Slot;
        Start_OK := True;
      or
        when Start_OK =>
          accept Start_Foreground;
          Start_OK := False;
      end select;
    end loop;
  end Foreground_Trigger;

package Foreground is
  function First_Ready return Integer;
    -- returns the number of the first ready foreground routine
    -- or zero if no routines are ready

  procedure Routine1;
    -- routine1 implementation

  procedure Routine2;
    -- routine2 implementation

  procedure Routine3;
    -- routine3 implementation

  procedure Routine4;
    -- routine4 implementation

end Foreground;


package body Foreground is separate;
```

Figure 2.7. Baseline Cyclic Executive with Foreground Processing (2 of 3)

```
task Foreground_Task is
  pragma priority(1);
end Foreground_Task;

task body Foreground_Task is

  begin
    loop -- forever
      Foreground_Trigger.Start_Foreground;        -- block so background runs
      while Foreground.First_Ready /= 0
      loop
        case Foreground.First_Ready of
          When 1 =>
            Foreground.Routine1;
          When 2 =>
            Foreground.Routine2;
          When 3 =>
            Foreground.Routine3;
          When 4 =>
            Foreground.Routine4;
        end case;
      end loop;
    end loop;
  end Foreground_Task;


task Background is
  pragma Priority(0);
end Background;

task body Background is
begin
  loop  -- forever
    -- Background Processing
  end loop;
end Background;
```

Figure 2.7. Baseline Cyclic Executive with Foreground Processing (3 of 3)

Baseline Executive - (Stop Overruns)
This executive includes low priority background processing, middle priority cyclic processing, and high priority event driven processing. The Baseline executive will differ from the other examples only in the area of periodic processing. The Baseline cyclic executive contains a minor cycle and a major cycle which is some multiple of the minor cycle. Each minor cycle within the major cycle has a set of routines to run (a frame). If the frame of routines does not complete in the minor cycle it is terminated and will be restarted at its next scheduled minor cycle.

Variation 1 - (Suspend Overruns)
This executive is the same as the baseline except instead of being terminated on overrun, the overrunning frame is suspended and is allowed to continue at the first available moment.

Variation 2 - (Log Overruns)
This executive is the same as the baseline except overrunning frames are allowed to continue overrunning, but the fact that they overran is logged.

Variation 3 - (Privileged Frame Overrun)
This executive is the same as the baseline except there are explicit rules about which minor frames can be overrun and which cannot be overrun.

Variation 4 - (Limited Consecutive Overrun)
This executive is the same as the baseline except that overruns are allowed as long as they are resolved in a certain length of time.

Variation 5 - (No Major Overrun)
This executive is the same as the baseline except overruns are permitted in the minor frames but not across a major frame boundary.

Variation 6 - (Sub-Slice Executive)
This executive is the same as the baseline except that the minor frame is subdivided into time slices in which the routines must finish or they will be terminated.

Table 2.1. Cyclic Executive Summary (1 of 2)

Variation 7 - (Foreground) This executive is the same as the baseline except that a foreground level has been added between cyclic processing and background processing.
Variation 8 - (Competing) This executive is the same as the baseline except that there are two cyclic structures of different minor cycle rates, competing for the same system resources.

Table 2.1. Cyclic Executive Summary (2 of 2)

2.3 APPLICATIONS OF THE CYCLIC EXECUTIVE

This section examines some of the applications for which the executives described might be used and reasons why they are considered suitable for such applications.

The baseline executive as described would be typical of a processor which was running several feedback control loops, where period and phase are critically important (feedback at the wrong time can be worse than no feedback at all). What also fits very well into this scheme are queued communication procedures. The inputs to these procedures could be either requests queued from other software modules or requests from interrupt handlers that the appropriate processing be done.

The suspension variation is typical of applications where software without "hard" periodic requirements has been placed in the cyclic executive. This often occurs when periodic processing and non-periodic processing are forced into the same procedures. A typical example would be a situation where data must be extracted from an input device at a precise period, then processed as soon as possible for output. The data extraction processing is periodic and by the statement of its requirement, one must assume that if it were to overrun its frame, the data retrieved would either be unavailable or worthless. The data processing, however, is not stated as a periodic requirement. The only requirement is that it process data as fast as possible with no dependence on either the period or the phase of the periodic processing. It is desirable for this processing to run in any available time. The only question is whether or not this processing belonged in a foreground process.

The overrun logging variation is useful in systems where periodic processing sections are much smaller than the time available for them to run, or where the periodic requirements are not "hard". The reason is that this structure tends to cause jitter any time a frame overruns. Each time a frame overruns, the start of the next frame is delayed until the previous frame completes. This situation not only causes a frame to start later than its scheduled time, but it also gives this frame less than its allotted time to complete before the subsequent frame is scheduled, making a subsequent overrun more likely. It might be argued that the executives described previously have some amount of jitter due to the overhead inherent in either the suspension or abortion of an executing frame. Although this is true in the cases discussed to this point, the jitter is controlled and an upper bound can be placed on it. Strongly periodic processing requires that the effects of potential jitter be analyzed, which requires that the maximum jitter be known. The overrun logging variation is a very commonly used scheduler for all sorts of simple applications where a low overhead, easily written, and easily understood executive is needed. When this variation is used it is necessary to control overrun in the design phase.

The privileged frame overrun variation represents a very common approach to overruns that cannot be completely controlled in the design phase. Executives belonging in this category have had their periodic and non-periodic requirements specifically analysed. Based on these requirements, specific rules are developed determining when overruns are allowed to occur and when they are not. These rules take into account the amount of variation in period and phase acceptable to each of the periodic routines, the amount of time non-periodic routines are likely to take, and what sort of compromises the specific software under consideration allows.

An example where this type of executive structure might be applied is as follows: A filter receives inputs at a variable rate but must output data to a feedback system at a 64hz rate. It is discovered that in cases where data input is faster than 128hz it is desirable to run the feedback at 128hz. The filter in some cases could finish fast enough to run at 128hz but in other cases could only run at 64hz. Given an executive that allowed one frame overrun coupled with feedback software which could run at either 128hz or 64hz depending on filter speed, this system could then run at 128hz whenever possible and drop to 64hz as necessary.

Variations 4 and 5 (Limited Consecutive Overrun and No Major Overrun) represent different methods of trading off overruns that cannot be completely controlled during the design phase.

The sub-slice variation might also be described as a special case of the privileged frame overrun executive, where minor cycle is the greatest common divisor of the time slices, and overruns are allowed within each time slice. An example is when several processes share a single processor. Each process normally fits into its time slice, but one or more of the processes have a recognized case where it will not fit into its time slice. Additionally, the probability of more than one of these cases occurring at any one time is exceedingly low. Finally, the system is better off having just one process fail to complete in the required time than to have all of the processes fail to complete. It seems advantageous to control overruns tightly such that one process overrun does not make the overrun of other processes more likely. This situation might be typical of a single computer controlling several different radars.

Adding a foreground to a cyclic executive gives the system designer the opportunity to place software with no periodic requirements in a structure better suited to those needs. Care must be taken at design time to assure that processing in the foreground will execute often enough to meet the system's timing requirements.

The competing cyclic structure variation is the most difficult to justify and to implement. When there are two competing cyclic structures the advantages of simplicity and transparency seem difficult to realize. It must be noted that in the nontrivial case, the minor frame periods must not only be different for the two structures, but they must also not be multiples of each other. In this case we have two processing structures synchronous within themselves, but asynchronous to each other. The only way this can be made into a simple problem is if one structure only has "soft" periodic requirements. In this case the structure with "hard" periodic requirements can have clear priority over the structure with "soft" requirements. If this is not possible the only other simple solution would be to combine the structures into a single cyclic executive running the software with slower periodic requirements at the faster rate. This approach may or may not work well but is very often done.

In this last case it is not even worth speculating on cyclic solutions where simple solutions fail. If the simple solutions fail, running two cyclic structures with no clear solution when they interfere defeats the entire purpose of a cyclic executive.

2.4 THE DATA DRIVEN DESIGN

For the purpose of comparison we introduce a more Ada-like time [illegible] design, which we will call a data driven design. A data [illegible] consists of a number of tasks which cooperate to perform the [illegible] the real time system. Scheduling is done by the run time [illegible] The next task to run will be selected from the set of tasks [illegible] on its scheduling criteria. The set of tasks ready [illegible] on the availablility of data within the system. It is assumed [illegible] recieves data through some number of rendezvous, does some [illegible] outputs data through some number of rendezvous. Those tasks [illegible] not received any data or have finished processing [illegible] blocked and not ready to run. Therefore only those tasks that [illegible] process will be ready to run. In this way a real time system [illegible] the data flow through the system to schedule tasks [illegible] time.

In the next several chapters the data driven design [illegible] to the cyclic executive techniques in handling [illegible] problems.

CHAPTER 3

TIMING PROBLEMS

Given the number of constraints that a given real-time system is required to satisfy, it is not surprising that most real-time systems cannot avoid timing errors. The general view among the community of real-time designers is to live in the face of such problems and apply engineering methods to minimize the impact of such problems. Most designers, unconciously, apply the principles of fault-tolerant system design to the problem of timing errors. This chapter discusses the issues of timing problems from a fault-tolerant perspective. The general view is that <u>timing faults</u> occur at various points which in turn lead to <u>timing errors</u> in related system components. These timing errors can lead to <u>timing failures</u> which if not averted can lead to disasters. As one of the major accomplishments of fault tolerant design illustrates, it is not necessary to isolate the causes of faults or faults themselves to provide satisfactory performance. What is necessary is to study and analyse the effects of errors on system components and formulate strategies to limit or recover from the effects of such errors.

This philosophy applies equally well to the problem of timing errors. In this chapter, we adopt the terminology and techniques of fault tolerant designs to study and characterize the problem of timing errors in real time systems. First, the basic concepts and terminology of fault tolerance will be reviewed. Then they will be used to describe the manner in which timing problems are treated in practice. In this process we note some of the similarities and differences among the two. For a detailed discussion of these concepts the reader is advised to consult references [4, 40].

3.1 BASIC CONCEPTS

<u>Faults</u> are undefined events which plague most of the current hardware and software systems. A fault occurring in a system may be due to design errors or to spontaneous events which may be caused by a hostile environment. The causes of faults are unimportant. What is important is the effect of the faults on system components and on system reliability. The occurrence of a

fault within a system component leads to an error or erroneous state. This state is in disagreement with the detailed system specifications. Any processing starting from an erroneous state will lead to a failure. Any processing starting from a failure will eventually lead to the system not meeting its external specifications.

The distinction between errors and failures is a bit fuzzy. An error state is an internal state of a component. This state is in general due to a fault and is not supposed to occur according to the detailed design of the component itself. If the error is not detected or not handled, it will cause subsequent errors in the component itself and/or other components of the system. An error in the external behavior of the component or of the entire system is termed a failure. It can be seen that the term failure applied to the component of a system is an error of the system, etc.

The reliability of a system refers to the probability of the occurrence of failures in the system. The various measures of reliability are:

[1] mean time between failures (MTBF),

[2] mean time to repair (MTTR),

[3] frequency of failure occurrence (actual or predicted),

[4] availability (the fraction of time a given service is provided)

The effects and severity of failures may also define the reliability of a system. If a critical component fails, even with high availability, it may lead to disaster.

3.2 FAULT TOLERANCE TECHNIQUES

The most common and widely used technique for fault-tolerance involves the following phases.

[1] Error Detection: This phase concerns the actual detection of an erroneous state in the system or some component thereof. Several techniques are used. The most commonly used technique is to perform assertion checks during the course of the computation. At certain points, a verification of the correct state is made. Another technique is to perform an acceptability test. This involves checking the inputs or the outputs of a system component. The input checks are called interface checks. They are executed to determine if the rest of the system is performing properly. The output tests may be in the form of checking certain or all of the outputs. Yet another form of error detection technique is to use redundancy.

Several components of identical specifications are asked to compute a certain result and, if the majority of the components produce differing results, there is an error.

[2] Damage Assessment: Once an error is detected, it is useful to classify the effects of the error or its potential effects on the rest of the system. Depending on the nature of the error it may be appropriate to perform some error recovery, or maybe no error recovery is needed. The classifications of an error are based on two attributes: its extent, and its incidence. The extent determines if the error is local or pervasive. For local errors, certain local error recovery may be performed, or some form of damage confinement may be necessary. A pervasive error usually has effects on other system components; therefore, any error recovery method must seek to correct or minimize the damage caused to rest of the system. The incidence of errors determines if the error is transient or persistent. A transient but local error may sometimes be ignored in the face of meeting real time deadlines. If the error becomes persistent, then it will severely affect the performance of the system. Thus, the system must maintain a history of errors occurring in the system to determine whether they are transient or persistent. Usually, if a persistent and pervasive error is encountered, then it may be necessary to shut down the system. If the error is persistent but local, then it may be appropriate to replace the faulty component or perhaps going back to some previous state.

[3] Error Recovery: The most commonly used techniques of error recovery are classified into two categories: backward error recovery and forward error recovery. Backward error recovery, essentially means taking the system back to some point previous to the error's occurance. This can be done locally or globally, depending on the interaction of the components and locality of the error. Forward error recovery techniques do just the opposite. They try to repair the error state as much as possible and establish a continuation point. This continuation point defines a state which is safe for a function that would be performed some time in the future under normal circumstances. In practice, however, this method can only be applied if the error situations are anticipated and a forward recovery point is known. In many real-time systems it would be better to perform forward error recovery.

3.3 REAL TIME FAULT TOLERANCE

In real time systems timing faults and errors can lead to a failure to meet the real-time requirements expressed in terms of deadlines and throughputs. The sources of timing faults can never really be tracked down during the development cycle. Thus the timing errors detected at runtime have to be handled and treated in real time as well. This is the only major conceptual difference between general fault tolerant strategies and those employed in real-time systems. The importance of such constraints should be obvious to the reader. Error detection, damage detection, and error recovery must be performed in real time. This requirement guides the design of fault tolerant strategies for timing as well as functional errors.

Most of the timing faults leading up to timing errors usually lead to a violation of deadline or throughput requirements. Before describing methods of dealing with deadline and throughput problems it is necessary to understand these requirements. We introduce some informal definitions to provide a better understanding of the issues involved.

For our purposes we define throughput as the number of inputs completely processed in a given amount of time. A throughput requirement would be violated if not enough data was processed to meet the needs of the particular application. Throughput may not be adequate when either there is an abnormally large number of inputs to be processed or an abnormally low number of inputs are being processed.

Deadlines are defined as points in time by which the software component must have reached a particular processing state. If it fails to meet a deadline the system will have failed to perform its mission. Deadlines are jeopardized when other high priority processing is scheduled too near the deadline.

Although these two problems are quite different, a timing fault that causes one problem is likely also to cause the other. A system that consistently misses deadlines is likely to have throughput problems and vice versa.

The way deadline and throughput problems are detected and handled has an impact on the capability of a real time system to meet its timing requirements. The detection and handling of these problems is largely dependent on the system structuring* used. The ability and the effectiveness of various strategies that try to resolve these problems differ not only among different scheduling disciplines but also between different variations of the same scheduling discipline.

The two scheduling disciplines of interest in this chapter are those called for in the cyclic executive design method and in the data driven design method.

*The term "system structuring" includes the concept of functional organization as well as the temporal organization or the scheduling policy.

3.4 TIMING ERROR DETECTION IN A CYCLIC EXECUTIVE

In a cyclic executive system an artificial schedule is generated such that if the system keeps the schedule, timing requirements will be met. This schedule is composed of frames each of which is given one minor cycle to complete. After all the frames have run the major cycle is complete, and the frames start over from the beginning again.

In a cyclic executive system there are three basic ways of detecting timing errors: overrun frames, overloaded queues, and missed deadlines. Overrun frames and overloaded queues represent timing errors. Missed deadlines represent timing failures. A frame overrun occurs when a frame has not finished executing by the time its minor cycle completes. It is detected by the mechanism for scheduling the new frame. An overrun frame is an important indication of potential timing problems in that cyclic executives are tuned so that no timing problems will occur as long as no frames overrun. Once a frame overruns, the basic assumptions on the timing behavior of the system are no longer valid. Of the timing problems that could occur after a missed frame, a missed deadline is the most likely. A deadline might be missed either because the deadlined software is in the overrunning frame and will be cut off, or because the deadline is in the next frame and will be delayed by the overrun. While a frame overrun indicates a possible timing problem, it does not isolate the cause of the problem because there is no indication as to which routine overran, or whether this is an isolated occurrence or a general trend. At best, a frame overrun can be regarded as an early warning that timing trouble may be coming.

Throughput is not as likely to be affected by one overrun frame. The relationship of overrun frames to throughput problems is much weaker than the relationship to missed deadlines. If a piece of software frequently overruns its frames, it is very likely that a throughput problem will develop. Not all throughput problems, however, are accompanied by overrun frames. This is especially true when routines are explicitly limited in how much input they can process in a major cycle.

An overloaded queue is another sign of impending timing danger. This occurs when a queue starts to fill up past the amount of data it is expected to contain. Generally, some extra space is set aside in a queue that is not supposed to be used in normal operation but can be used if necessary. When this area is being used it means the queue is overloaded and that there is a throughput problem; the software that is supposed to empty this queue does not have a high enough throughput to handle the current load. This situation presents two dangers. First, as the queue fills, the time it takes for the system to respond to a new input increases, jeopardizing the system response requirement. Second, if the queue keeps filling, eventually it will overflow, and data will be lost.

Missed deadlines differ from frame overruns in that a frame overrun indicates the violation of an artificial constraint generated to aid the designer in timing analysis and verification - an error has occurred, not a failure. When a deadline is missed it means that a timing failure has occurred. The effect of this failure depends on the application. Some applications may require complete system shutdown and restart; others may not require such drastic action. To detect a missed deadline, several pieces of information are needed: the deadline, the amount of time the by which software is allowed to miss the deadline, and the time at which the required action was performed. From this information and a knowledge of the application, the severity of the overrun can be determined and an appropriate course of action chosen. The utility of detecting these missed deadlines is twofold. First, during the executive tuning process these overruns can be addressed by a number of tuning mechanisms. Second, during operation a missed deadline can have serious performance repercussions. Often action must be taken to prevent serious damage from being inflicted on the system's hardware. An example is a feedback loop that controls the reactions inside a nuclear reactor. A missed deadline could destabilize the filter used to keep the reaction within safe levels. In this case, a missed deadline might be a good reason to trigger a reactor shutdown in order to avoid serious consequences.

Of these three indications of overrun/overload conditions, two are early warnings that timing problems might occur, and one is an indication that a timing problem has occurred. Although we speak of frame overruns and overloaded queues as early warning signs, there is no guarantee that a timing problem has not already occurred by the time one of these indications occurs. (Processing that has very strict response requirements may have already overrun its response deadline by the time the frame overruns or the next element is ready for the input queue.) A single frame overrun indication has a very different meaning than an overloaded queue indication. The frame overrun indicates that the conditions are ripe for a missed deadline. The overloaded queue indicates that system performance is in jeopardy due to an excess of data or a lack of throughput. A frame overrun is a momentary indication with no specific cause indicated. An overloaded queue is an indication of the general trend of processing and indicates specific trouble spots.

3.5 DAMAGE ASSESSMENT AND ERROR RECOVERY IN A CYCLIC EXECUTIVE

There are many responses possible when timing difficulties threaten, depending on the type of trouble indication. In this section we discuss the ways timing problems are handled in a cyclic executive. These methods are summarized in Appendix A.

When a frame overrun occurs the application program must brace itself for an unknown missed deadline. Depending on the application there may or may not be a way to do this (e.g., in a critical feedback loop, the feedback gain might be adjusted momentarily to prevent the destabilization of the loop). Perhaps the most helpful aspect of frame overrun detection is that during the executive tuning process these overruns indicate potential timing problems to be addressed. Although these indications do not help at the time of occurrence, they do help to avoid future occurrences. Frame overrun detection

3.4 TIMING ERROR DETECTION IN A CYCLIC EXECUTIVE

In a cyclic executive system an artificial schedule is generated such that if the system keeps the schedule, timing requirements will be met. This schedule is composed of frames each of which is given one minor cycle to complete. After all the frames have run the major cycle is complete, and the frames start over from the beginning again.

In a cyclic executive system there are three basic ways of detecting timing errors: overrun frames, overloaded queues, and missed deadlines. Overrun frames and overloaded queues represent timing errors. Missed deadlines represent timing failures. A frame overrun occurs when a frame has not finished executing by the time its minor cycle completes. It is detected by the mechanism for scheduling the new frame. An overrun frame is an important indication of potential timing problems in that cyclic executives are tuned so that no timing problems will occur as long as no frames overrun. Once a frame overruns, the basic assumptions on the timing behavior of the system are no longer valid. Of the timing problems that could occur after a missed frame, a missed deadline is the most likely. A deadline might be missed either because the deadlined software is in the overrunning frame and will be cut off, or because the deadline is in the next frame and will be delayed by the overrun. While a frame overrun indicates a possible timing problem, it does not isolate the cause of the problem because there is no indication as to which routine overran, or whether this is an isolated occurrence or a general trend. At best, a frame overrun can be regarded as an early warning that timing trouble may be coming.

Throughput is not as likely to be affected by one overrun frame. The relationship of overrun frames to throughput problems is much weaker than the relationship to missed deadlines. If a piece of software frequently overruns its frames, it is very likely that a throughput problem will develop. Not all throughput problems, however, are accompanied by overrun frames. This is especially true when routines are explicitly limited in how much input they can process in a major cycle.

An overloaded queue is another sign of impending timing danger. This occurs when a queue starts to fill up past the amount of data it is expected to contain. Generally, some extra space is set aside in a queue that is not supposed to be used in normal operation but can be used if necessary. When this area is being used it means the queue is overloaded and that there is a throughput problem; the software that is supposed to empty this queue does not have a high enough throughput to handle the current load. This situation presents two dangers. First, as the queue fills, the time it takes for the system to respond to a new input increases, jeopardizing the system response requirement. Second, if the queue keeps filling, eventually it will overflow, and data will be lost.

Missed deadlines differ from frame overruns in that a frame overrun indicates the violation of an artificial constraint generated to aid the designer in allows the designer to detect inaccuracies in the timing estimates upon which the initial executive tuning was based. Cases where important timing interactions were overlooked are likely to be spotted as frame overruns.

The use of frame overruns to detect timing analysis problems is key to the concept of using cyclic executives. When a designer uses a cyclic executive he simplifies the task of verifying that timing constraints will be met. This simplification is accomplished by first verifying that if no frame overruns, then timing requirements will be met. Software is then assigned to frames such that frame overruns will not occur. When frame overruns do occur they indicate that timing assumptions have been violated. The investigation of the causes and effects of any possible frame overrun becomes crucially important to the success of the cyclic system.

Overloaded queue indications can be handled either by a reduction in the amount of data entering the system or by an increase in the amount of processor time available for critical processing. A strategy for dealing with timing must be developed depending on the requirements of the application. Several possibilities exist to combat throughput problems. The external system can be adjusted to provide less data (e.g., a radar receiver's sensitivity can be lowered). Tests can be introduced to reject less important data (e.g., a radar may only process inbound targets). The time line of the cyclic executive can be adjusted so that less important routines are called less often to give critical routines more time (the rate of update to a secondary display can be cut to provide more time for tactical functions).

There are two crucial assumptions made in approaching any problem with the cyclic executive approach. First, the designer must assume that the timing requirements of any software to be handled in the cyclic executive can be reduced to periodic requirements at the period of the minor cycle or a derivative period. Second, the designer must be able to manipulate the relative phase of this processing in order to spread the processing out within the major cycle. This means the designer must be able to change the period of any periodic processing with an undesirable period and must be able to reduce response requirements to periodic requirements. Any processing for which this is not possible cannot be handled in a cyclic executive.

Changing the period of a periodic requirement is an easily understandable process. Most periodic processing can be modified to run at a rate faster than intended with no loss of performance. The design problem becomes simply choosing a period that fits into the cycle and is faster than the period specified.

Changing response requirements to periodic requirements is far more difficult. A response deadline typically is specified such that some processing state must be achieved within a certain amount of time after some external stimulus. In order for this type of requirement to be translated into a cyclic schedule two things are necessary. First, the schedule of the external stimulus must be known prior to run time. Second, the schedule of the external stimulus must be a derivative rate of the minor cycle. If these conditions are met the processing to respond to the stimulus can be scheduled by the cyclic executive. Otherwise, this processing must be handled outside the cyclic schedule.

Handling a significant amount of processing outside the cyclic schedule can cause serious problem for cyclic executive systems. This is best illustrated by an example. Imagine a system with one event driven process outside the cyclic executive. This event driven process has a very strict response requirement and occurs approximately every 1000 minor cycles but its period varies. The event driven process consumes processing time equivalent to 30% of a minor cycle. In order for the designer to guarantee that no frame will overrun he must limit each frame's processing time to 70% of a minor cycle just in case the event driven process interrupts that frame. The result is that the system processing capacity has been reduced by 30%. Additionally, the start time of periodic routines can be changed by up to 30% of a minor cycle. This, of course, jeopardizes the system jitter requirements (a jitter requirement specifies how much a routine may vary from a periodic schedule). Obviously, processing of this sort must be held to a minimum in a cyclic executive system, if the system is to succeed in meeting its timing requirements.

3.6 TIMING ERROR DETECTION IN A DATA DRIVEN DESIGN

Real time design in a data driven system is an attempt to map timing requirements to software without the cyclic executive crutch. This creates a different method of detecting and dealing with timing problems. There is no equivalent of a frame in a data driven design and therefore no frame overrun. This leaves the system designer with only two of the timing problem indicators of the cyclic system: filling queues and missed deadlines. The data driven design technique automatically allocates time to high priority routines before low priority routines; therefore, timing problems will usually show up in low priority software before high priority software is affected. The warning given by the struggling of low priority software, however, is similar in effect to the overloaded queue indication. It indicates a general ongoing throughput problem. There is no run time indication when two pieces of high priority software have to run at the same time in order to meet their timing requirements. The reason for this is that it is not an error when software competes in the data driven design; it is expected to occur. The error occurs only when this competition prevents the software from meeting system timing requirements.

There is also a difference in the type of timing problems that a designer is concerned about in the data driven design. Because time is automatically allocated to the highest priority routines as they need it, throughput is not as significant a worry in a data driven design as in a cyclic executive. If there is a throughput problem the only solutions are to use a faster processor, make the code more efficient, or cut down on the amount of input data (time has already been allocated as well as possible). The problem of missing deadlines, however, is even more of a problem than in the cyclic executive system. In a cyclic system high priority routines are kept separate by being assigned to different frames. This means they are not likely to interfere with each other. In a data driven design there is no mechanism to assure this. Two routines may need to be scheduled at the same time, both with deadlines that will be missed if they are not scheduled.

The problem is that the method of detecting timing problems is geared towards throughput problems, but the problems that are the most serious are not caused by inadequate throughput but potential interference between high priority routines. If there has been a careful timing analysis and tuning of the data driven design, this type of interference should only occur when the timing assumptions have been violated. This could be easily checked by inserting code to check critical timing assumptions. The problem with this approach is that there is no well defined method for analyzing and tuning a data driven design. A process to accomplish this is theoretically possible but it remains to be seen if a practical method can be developed.

3.7 DAMAGE ASSESSMENT AND ERROR RECOVERY IN A DATA DRIVEN DESIGN

Handling timing problems in a data driven design is primarily a matter of defining a framework in which system timing analysis and verification of timing requirements can be performed. There are many possible ways this type of framework could be built. In this section we discuss some ideas that might be useful in approaching the problem. Appendix A summmarizes some of the important ideas. The data driven design has the advantage of allowing the real time designer to pick the timing analysis and verification methods most appropriate to the application rather than limiting him to a single method. This advantage can only be realized, however, when these methods have been fully developed and proven effective.

A framework for the timing analysis and verification of real time requirements in a data driven system might be built on the basic assumptions described as part of a cyclic design. If the timing requirements of all the significant processing in the system can be reduced to periodic requirements with the same base period, and the phase of the system inputs within the major cycle can be controlled, then the control of the inputs can be used to spread high priority processing out across an input cycle. By controlling the schedule of system inputs, many advantages of the cyclic executive can be synthesized in a data driven system. The assumptions about the problem which allow this approach are no more restrictive than the assumptions that must be made before a cyclic executive system can be developed. The result is a very conservative framework in which to develop a method for timing analysis and verification for the data driven design. There are other less restrictive approaches that can be taken. Finding these approaches is an area requiring further research.

In general a framework for solving the timing problems in a data driven system must first detect potential interference and then limit that interference so that it cannot cause a timing problem. Whatever approach is taken it must rely on timing analysis techniques to help find potential conflicts (discussed in chapter 8), and tuning mechanisms to eliminate the possiblity of a conflict once it is found (discussed in chapter 7).

Once a timing problem is found in a data driven design there is a limited number of things that can be done. A throughput problem might be handled by thinning the number of inputs to the system, or by changing to a more efficient mode (the next chapter discusses such mode changes). This may not avoid the immediate problem, but might stop it from becoming a failure later on.

When a deadline problem is found in a data driven system there is not much that can be done unless the problem was anticipated by the designer. If the designer has anticipated the problem there may be a way of dummying up an output to avoid harming the system, otherwise the error recovery will have to be made at the next level of the system.

The major problem in terms of deadline problems in a data driven design system is that they are unlikely to be found before they have already resulted in a failure. The only way to improve this situation would be to insert code to check the validity of the timing model that was used to tune the system. Depending on the model used, this could be difficult. This is an area that requires further investigation.

3.8 COMPARISON

A comparison of the two scheduling techniques' ability to detect and handle deadline and throughput problems must be addressed on two levels, the ability of the technique to discover and address these problems during development, and the ability to discover and address these problems during operation. Further, the problems must be divided into two categories, those that are related to throughput problems, and those that are due to momentary effects such as interference between routines.

During the design of the cyclic executive there are well defined techniques for discovering and solving timing problems of both the throughput and momentary types. This process is often referred to as tuning the cyclic executive. There is no such well-defined process for data driven design systems. Ward [46] points out that most efforts to achieve proper timing in these systems are trial and error processes. There is, however, potential for developing such methods. More work is needed in this area.

During run time the data driven design automatically adjusts the software schedule to maximize the throughput of high priority routines. This can provide significant throughput advantages over a cyclic executive, especially in cyclic systems where part of the software is idle while the rest is heavily loaded. On the other hand a cyclic executive is tuned to avoid interference between high priority routines. Any interference that does occur will be recorded as a frame overrun to be dealt with in later tuning efforts if necessary. The data driven design has no mechanism to differentiate between routine interference and normal routine competition. Interference will not be detected until after a deadline has been missed. By then the original cause of the interference may not be discernible.

3.9 OTHER ISSUES

A great deal of attention has been paid to specialized scheduling schemes for real time systems. These schemes would take software such as that from a data driven design and schedule it in an optimum manner. Optimum performance is traditionally defined in such systems as a scheduler that, given a set of processing to be accomplished and a deadline for each member of the set, schedules all processing such that all deadlines will be met if they could be met by any scheduling algorithm. Depending on how many processors comprise the target system there are various algorithms to accomplish this [29]. This approach to real time scheduling is not adequate to meet the timing requirements of many real time systems. What is generally ignored in the definition of these schedulers is how it is determined when a piece of software should become eligible to run and how the deadlines are fixed. There must be a pre-run time separation of critical processing elements to guarantee that the run time scheduler will not be presented with a situation where no run time scheduler can succeed. The cyclic executive scheduling system, while generally not optimal at run time, uses this pre-run time definition scheme to guarantee adequate run time performance.

It is important to realize that the ability to anticipate the run time characteristics of a system is not limited to cyclic executive systems. While data driven systems are inherently more complex than cyclic systems, they can be deterministically traced as long as they are driven by a deterministic scheduler. This means their timing performance can be predicted ahead of time. In addition, there exist many mechanisms to simplify the timing performance of critical processing elements. These methods range from design time or run time control over the inputs schedule to the implementation of competing processing elements as co-routines.

In many applications it is possible to control the schedule of when routines become ready to run and when their deadlines occur. This is the property that allows cyclic executives to exist. An example might be a situation where a processor controls the position of two radar antennae. The position of each antenna must be corrected every 32 milliseconds. Although each correction to the same radar antenna must be 32 milliseconds apart, there is no requirements for how far apart the corrections to different antennae might be. Cyclic executives can be successful because processing can be evenly distributed at design time across a cycle through the manipulation of program start times and deadlines within that cycle. In this example a cyclic executive might schedule antenna corrections to different radars 16 milliseconds apart. The same type of distribution can be accomplished in the data driven design.

Many systems provide a way for the program to control the schedule of data input within the basic cycle. This usually takes the form of a special initialization procedure or of software polling. In the initialization case, the software starts up the external devices according to a specific schedule. A truly periodic device will deliver data periodically from the time of its startup. By controlling the phase of the startup the software controls the phase of all data deliveries. In the polling case, the software is responsible for initiating external data transfers at the right time. In either case the system designer can use his control of the input schedule to spread critical processing out over the system cycle and to avoid scheduling interference between high priority routines. This technique can be viewed as controlling the relative phase of the software as well as the period. These mechanisms are equally useful for the data driven designer and the cyclic executive designer.

Control of data propagation can also be used to achieve timing goals. Limiting the propagation of data in circumstances where a routine might fire and interfere with critical processing will prevent many problems. This control could be accomplished by a buffer task that checks the state of other tasks in the system before releasing data.

Graceful degradation is a critical requirement for many real time systems. This is especially true in processing overload situations. It is crucial for the system to be able to shift processor resources from low priority processing to high priority processing when necessary. The data driven design automatically performs this shift. The cyclic executive system must introduce a series of mode changes to accomplish the same thing. The data driven design has the advantage of reacting to each change in processor load as needed. The cyclic executive can only react on major shifts in processor load but provides more control over the exact allocation of the processor resource.

The processor is not the only critical resource in many systems. For the purposes of this chapter a critical resource is any resource used by more than one process where the resource is not re-entrant. These resources can become bottlenecks in many systems, as when several processes are waiting on this one resource and nothing is ready to run. To avoid wasted time due to bottlenecks, care must be taken to spread out the processing that requires each critical resource.

3.10 SUMMARY

The cyclic executive scheduling system provides a method by which software can be allocated to specific time slots according to timing estimates. The resulting system can then be evaluated as to how well these timing estimates predicted system performance, and the system can be tuned to remove any performance glitches. This technique provides an intermediate step to achieve the real system timing requirements.

The data driven design provides many opportunities to perform the same type of system tuning. As yet, however, there is no well-defined method for approaching the analysis and tuning of a data driven system. The data driven design, however, has some superior performance characteristics such as high processor usage and consequently high throughput.

The development of timing analysis and tuning methods for data driven designs would provide many benefits for the real time designer ranging from higher throughputs to easier maintenance. The development of this method faces many challenges in the form of analysis complexity.

CHAPTER 4

MODE CHANGES

Mode changing is a process whereby the function of the overall system is altered to include new functions, drop old functions, and/or change the resource allocation among the functions in the system. Two examples might be a radar switching from acquisition mode to track mode, or a communication cell switching from receive mode to transmit mode. The abstract components of a mode change are shown in Figure 4.1. Some reasons one might want to invoke a mode change include: to invoke a new set of functions that process a completely new set of data, to invoke a new set of functions which handle the same data as the old functions but perform different operations on them, or to change resource allocation or data flow among existing functions.

The abstract components of the mode change shown in Figure 4.1 can be represented in many different ways depending on the scheduling mechanism used. Several of these components may be combined into one operation in some scheduling systems. The first two components listed involve turning off the inputs and outputs of the old mode and turning on the inputs and outputs of the new mode, setting the stage for the new mode's processing. The next two steps disable the processing elements of the old mode and enable the processing elements of the new mode. Once processing resources are allocated to the new mode, it will be ready to run. The next step is to redefine the criteria by which processing resources are allocated to processing elements in order to ensure that the processing elements of the new mode will be allocated processing resources according to the priorities of the new mode. Finally, the criteria for deciding that a timing fault has occurred must be changed to fit the timing requirements of the new mode.

The Components of a Mode change:

- o Disable Old Inputs and Outputs
- o Enable New Inputs and Outputs
- o Disable Old Algorithms
- o Enable New Algorithms
- o Replace Resource Allocation Criteria
- o Replace Timing Fault Declaration Criteria

Figure 4.1. Abstract Mode Change

Mode changes are initiated in one of three ways: operator initiation, application initiation, or scheduler initiation. The difference lies in who decides a mode change is necessary and for what reason. Operator initiated mode changes occur when the system operator detects a reason to change the function of the system (e.g., a pilot might wish to change his flight path). Application initiated mode changes occur when the application program detects a situation that requires a change in system mission (e.g., a radar may transition to track mode when a target has been detected). Scheduler initiated mode changes occur when the mode has to be changed to improve the timing characteristics of the system (e.g., secondary functions might be slowed or eliminated to improve tactical software throughput). Operator and application initiated mode changes tend to be composed of all six components of the mode change model shown in Figure 4.1. Scheduler initiated mode changes tend to be primarily changes to resource allocation criteria and timing fault declaration criteria.

The method of changing mode is heavily dependent on the scheduling methodology of the system. For each real time scheduling method there is a different set of reasons to invoke a mode change and a different way to accomplish it.

This chapter will discuss the issues of mode changing in the context of two real time scheduling methods: cyclic scheduling and data driven scheduling. In each scheduling method the reasons to invoke a mode change and the way the mode change is accomplished will be different.

4.1 MODE CHANGES IN CYCLIC SCHEDULING SYSTEMS

In a cyclic executive system mode changes are used for all of the purposes described above. A radical change in function is one of the primary reasons to implement a mode change in a cyclic executive system, but others are just as important. A complete change in the timing structure of the software is often needed to adapt the system to changes in the system's environment, even without real changes in the system's function. There may be a change in the priorities of some of the functions in the system which requires that current time allocation be adjusted to suit the needs of the functions that are currently most important.

In a cyclic executive system several approaches can be used to implement a mode change. Often a routine will incorporate logic to detect a mode change and adjust its function to fit the needs of the new environment. If timing changes are not too severe, the cyclic executive can adopt a new set of frame assignments. When major changes in timing are necessary a whole new cyclic structure may have to be adopted. Examples of code segments using each of these approaches are shown in Appendix B. For each mode change there are two considerations. First, the routines in the new mode must have an assigned slot in the cyclic schedule. Second, the data flow of the new mode must be established. The first is accomplished through modification of the cyclic scheduler either by changing the frame assignments to fit the new mode or by replacing the entire schedule with a new one designed for the new mode. The data flow is modified either by routines that adjust to the new mode or by new routines that can access the data they need directly.

Routines that change to fit the new mode usually fulfill similar functional requirements in both modes. Routines representing functional requirements that are unique to a mode are usually turned on and off with the changing of the mode.

Introducing new frame assignments into an existing cyclic executive allows the system to change the timing of routines that are common to both modes, and to introduce routines unique to the new mode while phasing out routines that are no longer required. This process usually involves the loss of one cycle's processing time before the new mode is ready to do useful work.

When timing changes so radically that a totally new cyclic structure has to be introduced, it usually indicates a change in the focus of the system or a total change in mission. A change in focus might involve running a small set of the original functions at a faster rate or a greater number of functions a slower rate. A new mission would involve scrapping all but the most basic of the original functions and substituting a whole new software system.

To this point we have only discussed mode changes in the cyclic part of a cyclic executive system. It is possible for a mode change to affect other parts of the system as well. We will discuss mode changes in three other parts of a cyclic system: event driven, foreground, and background. As the name implies, event driven software runs according to the dictates of external stimuli. Changing mode in event driven routines usually is just a response to a changing external environment. This type of mode changing requires no action on the part of the software. The change happens automatically in response to a different set of external stimuli.

Foreground mode changes tend to be very similar to data driven mode changes (see section 4.2). In fact the foreground can often be compared to a data driven design involving only part of the processor.

Background mode changes involve calling a different set of background routines. Background processing can be thought of as a loop containing calls to a set of background routines. Background routines always run to completion. A mode change can be thought of as a second loop containing calls with different routines in it. Mode transitions occur after the completion of the routine during which the mode change was commanded.

4.2 MODE CHANGES IN DATA DRIVEN SYSTEMS

In a data driven design adjustments to handle changes in system timing are done automatically as part of the scheduler, without the need for a mode change. Thus mode changes in a data driven system tend to be changes in function initiated either by the operator or the application program, not scheduler initiated changes to improve timing performance. There are, however, some mode changes to improve timing performance. These usually involve changing the focus of the system by removing parts of the data flow that are less important to the current mode.

In a data driven design, scheduling is a function of system data flow. Therefore, a mode change can be accomplished through redirection of data flow. Data is directed away from functions that are to be phased out and towards functions that are to be phased in. Functions whose role is redefined by the mode change may find themselves in a completely different part of the data stream after a mode change. The scheduling of the new mode is changed by the data as it flows through the new data paths.

Each type of mode change is supported by different mode changing techniques in the data driven design. Examples of code segments paralleling the cyclic cases are shown in Appendix B. The data driven design has a great deal more flexibility in methods for changing modes than the cyclic executive. Any change in the data flow can be regarded as a mode change. There are many ways of changing the data flow. One very convenient way of changing data flow is to use buffer tasks. Buffer tasks present an ideal method of manipulating the data in a system. Buffer tasks can select different inputs or different outputs based on mode. Buffer tasks can also rid the system of unwanted data upon a mode change either by flushing queues of data left over from the previous mode or by not accepting new data that is not needed for the new mode.

The cyclic case of a routine changing its function when it detects a change in mode can be represented in a data driven design by a similar structure, a task that changes its function when a mode change occurs. The detection of a mode change can either be accomplished by sending the task a copy of the current mode or by sending it a more specific signal indicating a specific mode change. Methods of disseminating mode information will be discussed later.

The type of mode change that is accomplished in the cyclic case by redefining the frames in the cyclic executive is accomplished in the data driven case by redirecting data outputs from the old functions to the new. The old function that does not receive any data will remain blocked and thus will not be scheduled. The new function that receives data will become unblocked and start executing its computation. The redirection of the data stream can be accomplished by informing key tasks of the current mode, for example placing data buffers at key points in the data flow and directing them to pass the data to the appropriate output task. The data buffers in the system then become the primary controllers of the mode. A buffer task that accomplishes this type of data passing is shown in Figure 4.2.

Another type of mode transition mentioned for the cyclic case, where the entire cyclic scheduler must be replaced, can be handled as a combination of redirection of existing data paths and multiple independent data flow architectures. A change of focus in mission can be accomplished by either shutting off the flow of data to part of the software or starting the flow into new software elements. A complete change in mission can be viewed as a completely separate data flow. When the mode is changed, data is cut off from the original data flow architecture and sent into the new architecture.

The data driven design can also make use of priority to affect subtly the mode of the process. By carefully selecting the priority of the tasks that are introduced and cut off from the data flow, the relative priority of various data paths can be significantly altered. This mechanism could be used to control what the system scheduler thinks is important at any given time. The ability to use priority to influence the time allocation of the system may be limited in some systems by the lack of dynamic priorities. It is quite possible for two particular processing elements to change in relative importance from one mode to the next, however, this difference cannot be represented with Ada's normal priority scheme.

4.3 COMPARISON

Each of the scheduling techniques makes use of its primary means of system scheduling to control system mode. The cyclic executive scheduling technique takes advantage of the fact that it can change the entire body of operating software by simply changing the scheduler. This approach centralizes the change so that it can be mostly accomplished by changing one function. The data driven technique takes advantage of the fact that its tasks are scheduled by the arrival of data. The purpose of the mode change is more easily understandable in terms of the overall system mission.

```
type Mode_Type is (Mode_1, Mode_2, Mode_3);

task Buffer_1 is
  entry Input(Input_Data : in Data_Type);
  entry Set_Mode(Input_Mode : in Mode_Type);
  entry Out_1(Output_Data : out Data_Type);
  entry Out_2(Output_Data : out Data_Type);
  entry Out_3(Output_Data : out Data_Type);
end Buffer_1;

task body Buffer_1 is
  Mode: Mode_Type := Mode_1;
  Data: Data_Type;
begin
  loop --  Forever
    select
      accept Input(Input_Data : in Data_Type) do
        Data := Input_Data;
      end Input;
    or
      accept Set_Mode(Input_Mode : in Mode_Type) do
        Mode := Input_Mode;
      end Mode;
    or
      when (Mode = Mode_1)  =>
        accept Out_1(Output_Data : out Data_Type) do
          Output_Data := Data;
        end Out_1;
    or
      when (Mode = Mode_2)  =>
        accept Out_2(Output_Data : out Data_Type) do
          Output_Data := Data;
        end Out_2;
    or
      when (Mode = Mode_3)  =>
        accept Out_3(Output_Data : out Data_Type) do
          Output_Data := Data;
        end Out_3;
    end select;
  end loop;
end Buffer_1;
```

Figure 4.2. Specialized Buffer Task -- Selects Output Based on Mode

MODE CHANGES

The major advantages of mode change in each scheduling technique parallel the advantages of the technique itself. The time at which the mode changes in the cyclic executive is very well defined - a major cycle will finish in one mode and start in the next. This precision may lead to problems in dealing with left over data and other transients of a sharp change. The data driven approach changes mode much more gradually, as the data from the adjusted routines spread through the system, minimizing the transients, but making it more difficult to determine exactly when the mode transition occurred. It is hard to determine if timing requirements placed on the mode transition were met.

4.4 OTHER ISSUES

There are several issues that are related to that of mode changing. These include loss of data, transients, overlays, mutually exclusive modes, dissemination of mode information, and degraded performance.

The loss of data varies in importance from application to application. The problem stems from the fast transition from one mode to another in a cyclic executive system. If a routine contains important data and that routine is no longer scheduled due to a mode change, that data is lost to the system until a further mode change occurs. Depending on what that data is, it could be worthless or it could be indispensable. For example the status of a routine that is not called in the new mode is relatively worthless, while the information that an inbound missile has been spotted would be invaluable in most modes. Steps can be taken to make sure important data is not lost. One approach is to keep this data in global data bases accessible to some function in the new mode that makes sure important data is passed on. This approach, while effective, leads to extra routines in the new mode, not only adding to its complexity, but also taking up valuable time.

The data driven technique does not suffer from loss of data, since routines continue to run as long as they have input data. This however can cause another problem. Data driven systems may experience problems with the amount of time it takes to make a complete transition to the new mode. If a task that is no longer useful in the new mode has a large queue of input data, it may continue operation until well into the new mode. If this mode was activated because of severe time pressure, the time taken up by this useless task can be a major problem. This problem can be addressed by instructing the queue not to feed the unwanted task in the new mode, or flushing the queue when the mode changes. This solution risks the problem of losing any important data that might be in the queue.

Quick mode changes in a cyclic system can cause problems other than just data loss. Often in real time systems the software is driving various hardware processes. The rate at which these processes are driven and the amount of time they take to complete can be very dependent on mode. When a mode is changed, there is no reason to assume that the hardware will be ready to start the first time it should be started in the new mode. It may well be finishing what the last mode had instructed it to do. The system must either be prepared to reset the hardware and re-initialize it from scratch, or it must wait for it to finish its last task before commencing operations in the new mode. In either case there must be a mode transition state in which the needs of the hardware are looked after. This can significantly lengthen the time it takes to change modes. Failure to accommodate the hardware can have serious repercussions, including expensive hardware failures.

In small systems, mode changes are used to conserve a limited memory space. This method identifies some mutually exclusive parts of different modes that can be treated as overlays. As the mode is changed the overlay part of the old mode is overwritten with the overlay part of the new mode. In the cyclic executive a mode changing state is introduced during which the overlaying process occurs. The disadvantage of this method is that the time taken to perform a mode change depends on the speed of the mass storage device used to store the software. This time can be minimized (if some extra memory is available) by pre-loading some subset of the new mode (identified as more crucial to the system mission) in the spare memory and creating a sub-mode in which this software runs alone while waiting for the rest of the software to be overlayed on top of the old mode.

The overlaying process in a data driven system is more complex for two reasons. First, there is no place in a data driven system where the components of each mode are listed and second, there is no set time when these functions start or stop running. In order to perform overlays in a data driven system a task must be introduced that knows the components of each mode and that can detect when they have stopped running due to a mode change. This task can then make that memory space available for overlays from the new mode. One method introduces a task which behaves like the memory system in a virtual memory machine. Provided there is memory to hold each mode this will consistently ensure that all the routines of the current mode are in memory. Unfortunately, this method of overlaying, in addition to being complex, can further stretch out the time necessary to make a mode change in a data driven system.

Sometimes there are relationships between modes (such as using the same hardware) that precludes running them simultaneously. In these situations the cyclic executive mode change works quite well, but not all data driven design mode change mechanisms work so well. There are ways around this problem. One way to handle the situation is to insert a task that blocks the new mode until the old mode is complete. The most effective way to enforce mutual exclusion, however, is to place the mutually exclusive code in the same task. The danger is that it could create an unacceptable situation in terms of modularity if the different modes are not functionally similar (i.e., similar inputs and outputs).

Dissemination of mode information is not a major problem in a cyclic executive system. Data can be shared in a cyclic system due to the inherent synchronization of the cyclic structure. Therefore a single mode variable can be kept for the whole system with no danger of overlapping access.

There are a number of possible schemes for dissemination of mode information in a data driven design. One of the more likely schemes is to keep mode in a package with access only through subprogram calls. This package could ensure that setting the mode could not overlap any other mode operation, while not limiting overlaps among mode read operations. (See Figure 4.3.) The disadvantage of this approach is that there is little control over where the mode change starts. In some systems it may be necessary that mode information be distributed in a specific order. To control the distribution there are several good approaches. There could be a task which calls mode entries of the various tasks in the system to distribute the mode in the correct order (Figure 4.4). Otherwise, mode could flow through the system just like any other piece of data (perhaps as part of a data structure that flows to all applicable tasks).

Graceful degradation of system performance is crucial in many applications. As load increases on mission critical software, it is often desirable to degrade the performance of less critical software to facilitate higher efficiency in critical areas (e.g., secondary display tasks might be slowed to prevent the loss of tactical data). Prioritizing tasks in a data driven design provides an exceptional method of accomplishing this degradation. It does not require a mode change of any kind -- processing power is automatically diverted to the primary system functions as needed. Cyclic executive systems must introduce a series of mode changes that successively allocate more and more of the major cycle to the primary system functions. On the one hand, it affords more control over the exact breakdown of time allocation. On the other hand, the software to produce this effect is more complex and cannot produce the continuous type of resource allocation available under the data driven design.

```
package body Mode_Holder is
  Mode: Mode_Type := Mode_1;

  task Write_Protector is
    entry Read_In;
    entry Read_Out;
    entry Write_In;
    entry Write_Out;
  end Write_Protector;

  procedure Read_Mode(Out_Mode : out Mode_Type) is
  begin
    Write_Protector.Read_In;
    Out_Mode := Mode;
    Write_Protector.Read_Out;
  end;

  procedure Write_Mode(In_Mode : in Mode_Type) is
  begin
    Write_Protector.Write_In;
    Mode := In_Mode;
    Write_Protector.Write_Out;
  end;
```

Figure 4.3. Mode Package (1 of 2)

```
  task body Write_Protector is
    Readers: Integer := 0;
    Writer: Boolean := False;
  begin
    loop  --  forever
      select
        when (not Writer) =>
          accept Read_In do
            Readers := Readers + 1;
          end Read_In;
      or
        when ((Readers = 0) and (not Writer)) =>
          accept Write_In do
            Writer := True;
          end Write_In;
      or
        accept Read_Out do
          Readers := Readers - 1;
        end Read_Out;
      or
        accept Write_Out do
          Writer := False;
        end Write_Out;
      end select;
    end loop;
  end Write_Protector;
end Mode_Holder;
```

Figure 4.3. Mode Package (2 of 2)

```
task body Mode_Distributor is
  Mode: Mode_Type := Mode_1;
begin
  loop --  Forever
    accept Set_Mode(New_Mode : in Mode_Type) do
      Mode := New_Mode;
    end Set_Mode;
    Buffer_1.Set_Mode(Mode);
    Buffer_2.Set_Mode(Mode);
    Buffer_3.Set_Mode(Mode);
    Buffer_4.Set_Mode(Mode);
  end loop;
end Mode_Distributor;
```

Figure 4.4. Mode Distribution Task

4.5 SUMMARY

We have seen how each scheduling mechanism adjusts the system mode in order to respond to changes in system mission. The major difference seems to be the speed in which the mode change is performed. In a cyclic executive the mode change is almost instantaneous. This fast change of mode can cause a number of problems that require mode changes to be lengthened artificially. The data driven design mode change is accomplished much more gradually, as data flows through the system. The mode change in a data driven design can be so gradual that it often is not recognized as a mode change at all, only as incremental adjustments to meet the current situation.

Appendix C contains a summary of some possible mode changing problems for each design method and suggestions on how they might be approached.

CHAPTER 5

AN EVALUATION OF CYCLIC APPROACHES

5.1 STRENGTHS AND WEAKNESSES OF CYCLIC EXECUTIVES

We now turn to the problems both solved and caused by the concept of the cyclic executive. First, we will describe in general the requirements that have led to the use of cyclic executives. Second, we will examine the problems the cyclic executive creates.

A cyclic executive is generally seen as a simple and efficient means of allocating the resources of an embedded real time system. It is often seen as a compromise between low overhead and multi-processing. It allows multiple processes to execute on the same machine using a well defined time line to allocate system resources. Overhead is minimized by a well determined number of process switches and the virtual elimination of scheduling decisions. Simplicity is maximized by precisely defining the time slot in which each process executes.

Processing performed in a cyclic executive system tends to have been placed there for one of two reasons. Either the processing is periodic in nature and must be performed at a precise rate, or the processing is really not periodic at all, but the cyclic executive provides a method of controlling the resources available to the processor.

The case of truly periodic processing clearly points to a cyclic executive for simplicity. When the various periodic scheduling requirements are combined in this fashion, not only is overhead reduced compared to timing each period independently, but also the synchronous structure of the executive will avoid unnecessary conflict over the allocation of system resources and will allow data to flow through the system in a consistent manner.

Non-periodic processing that is placed in the framework of a cyclic executive is usually one of two types: event driven processing delayed until its allocated time slot or processing which must be executed repeatedly, but not necessarily at any consistent rate.

Event driven communication has an appealing sense of immediacy in the way communication corresponds to the environment's needs. Unfortunately, a completely event driven executive often does not work in such a well defined manner. Because the processor is waiting for the external environment to tell it what to do, there is no guarantee that requests for processing will come in any sort of reasonable fashion. The result is a total lack of synchronization between the various software processes, a state which is often unacceptable in terms of data coordination and resource allocation. Loss of data integrity and spot overloading of the processor's resources could be the result of such a system. In many cases these problems can be avoided by placing much of the processing, otherwise driven by events, in a cyclic scheduling system and having the cyclic structure control the order of execution in a strictly synchronous fashion. The situation in this case has improved not only by alleviating the above mentioned problems but also by allowing the executive to detect other similar problems (ie. frame overruns) and make intelligent compromises when they occur.

Processing that must be repeated, though not at any rigid rate, falls into two groups: either it has a "soft" periodic requirement, or no real periodic requirement but instead a maximum latency requirement. A "soft" periodic requirement implies that although there is no requirement for strict periodicity, the average period should be a constant. Routines with no periodic requirements at all have to be run repeatedly but require neither a fixed periodic execution nor an average periodic execution. Typically the only timing requirement for these routines is a maximum time between run times; this maximum is usually far in excess of the expected period of the routine.

Another problem addressed by cyclic executives is analysis of system timing performance. This type of analysis can be performed quite simply in a system designed with a cyclic executive. Each program has access to the entire processor resource as it runs. Timing requirements are simply that all routines scheduled in a minor cycle complete execution within that minor cycle. This requirement can be verified simply by adding the execution times of the routines in each minor cycle together and adding in a maximum interference expected from asynchronous processing, thus yielding maximum expected execution time. This time can then be compared to the maximum time allowed. It is important to differentiate between simple timing analysis and either the system tuning process or the analysis of relative timing problems in the system itself, both of which will be discussed later. There are a number of problems that can creep into these other processes (such as routines that do not fit into a single frame) that must be addressed with more complex analysis of design tradeoffs. The simplicity of the basic timing analysis only provides a tool for addressing these problems.

Although the cyclic executive approach has a great deal of power in dealing with real time problems, it is not without its disadvantages. The disadvantages of cyclic executives can be classified in three categories: design problems, run time problems, and maintenance problems. Through the years tradeoffs have been developed to solve the design and run time problems; the result of these tradeoffs has been to further complicate the maintenance problem.

AN EVALUATION OF CYCLIC APPROACHES

The design problems inherent in cyclic executive design involve the cost of tuning the executive to obtain the desired level of performance. Even when accurate timing information is available about each of the routines to be called by the executive, the process of getting the timing right can be a nerve wracking, time consuming experience. The final configuration is often arrived at through a lengthy trial and error process. Even after the process is complete, it is difficult to say whether the executive is properly tuned or not. The number of variables associated with a system wide timing effort can be enormous and errors may not show up for years. Even if the timing is correct, it is difficult to document just how the timing configuration was reached. Because of the difficulty of this process, it is often viewed as an expensive form of "black magic" with which, once complete, no one dares to interfere.

Related to the tuning problem is the question of what to do with processing that does not fit into a minor cycle. One approach is to dissect the big routine into several little routines and run them in consecutive minor cycles. This approach not only leads to poor readability in the resulting code but also often causes a loss of efficiency in the code. Dissection even in its best cases causes problems, but in the worst cases may not even be possible. Some software cannot be divided. In these cases a pure cyclic structure cannot work. Some modification of the cyclic structure may yield a working executive, but this type of modification causes a break in the cyclic structure and further complicates all the problems associated with the design and maintenance of the executive.

Run time problems with cyclic executives come in three areas: flexibility, efficiency, and jitter. It is important to note that two of these areas (efficiency and jitter) are precisely the problems the cyclic executive is supposed to solve.

It is no surprise that a cyclic executive lacks flexibility. The whole concept of the cyclic model is to lock processing into a frozen time line. The problem is that events do not always happen in the same pattern. The cyclic executive has no ability to adjust the allocation of system resources to reflect the needs of the situation. When the need for adjustment becomes extreme, system designers often introduce a mode change to accomplish it (a mode change in this type of system involves replacing the current frame assignments with a new set of frame assignments). The lack of flexibility both causes the resulting system to be unable to accomplish its task efficiently in some cases and in extreme cases, requires additional software to be generated to satisfy the flexibility requirement.

A related problem is a potential difficulty in providing efficient frame assignments. This situation occurs when every routine fits into a minor cycle, but the routines do not fit together well into frames of the right size (consider the case where every routine takes 55% of a minor cycle). The system designer might address this at design time, trying to slice some of the routines into little pieces to make an efficient frame assignment, causing a design and maintenance nightmare, or the designer might accept the low efficiency of poor frame assignments.

A routine that has been placed at the end of the call list for a particular frame has a tremendous propensity for jitter (jitter occurs when the start time of periodic processing deviates from its precise requirement in an unpredictable manner). If any of the routines ahead of it on the call list has a variable execution time, jitter will occur. Frame assignment is again critical. Routines that cannot accept jitter must be placed before any routines of variable length execution. It is not always possible to make assignments this way and routines may have to be moved to other frames, causing other problems in the tuning of the e'ecutive.

We have examined many problems caused by cyclic executives during design and run time, but by far the most significant problems with cyclic executives are maintenance problems.

Although the cyclic scheduling code itself is often easy to read and quite simple, the routines to be scheduled are often rendered all but illegible by being cut in pieces and manipulated to fit into the cyclic executive's scheduling pattern. In addition, a large part of the design of the cyclic executive lies in selection of the proper scheduling pattern, but information about the criteria used in selecting this pattern is not generally part of the cyclic executive and is not documented. From a readability (or documentation) point of view alone, modification of a system designed with a cyclic executive is tough.

Readability is not the only maintenance problem with cyclic executives. The cyclic executive system is very fragile when code changes are made. Any changes to any routine in a cyclic system can change the system timing. The result of a change in timing could be any one of the run time problems discussed above. Bugs resulting from a change can surface in areas unassociated with the routine that was changed. Bugs may not appear for some time, leaving doubt as to which of several changes caused the bug. It may be necessary to retune the entire system every time a change is made.

Before leaving the area of maintenance it might be useful to envision a system design where several of the major components have been sliced into several routines apiece. You have been handed a list of requirements that must be added to one of these components. You are now faced with the task of determining which of the routines that make up this component performs the functions in which you are interested, then figure out how this routine interfaces with all the other routines that comprise the component, then decide if there is time in the routine to perform your new functions, and then study how the new functions might be implemented. If you run into a problem at any one of these steps, it may be necessary to reconstruct the component from its pieces and repartition it into a whole new set of routines.

5.2 SUMMARY

In conclusion, the performance problems inherent in the cyclic executive design methodology can generally be addressed and solved during the design of a particular system. The cost, in terms of design effort and maintainability of the system, can be enormous. Table 5.1 lists the problem areas addressed by cyclic executives and Table 5.2 shows the problems caused by cyclic executives.

Provides a low overhead method of achieving multiprocessing.
Allows precise periodic execution of software which requires periodic processing.
Provides a synchronous system where requirements for data synchronization can be relaxed due to implicit routine synchronization.
Helps to avert problems caused by events coming in bursts or an inconvenient order. Much of the processing related to event driven requirements can be assigned to a synchronous time slot. This approach also helps protect the data integrity of the event driven software.
Provides a framework in which system overloads are detected very quickly so action can be taken to limit the impact of the overload.
Provides a system in which performance requirements and actual software performance can be estimated very easily. These performance requirements include maximum execution time, jitter and maximum time between executions.

Table 5.1. Problems Addressed by Cyclic Executives.

The design cost of tuning the cyclic process to achieve the best possible time line is high.

Routines that do not fit into a minor frame must be divided into pieces which do, if possible. This causes less efficient code and loss of readability and leads to possible maintenance problems.

Run time problems:

```
    Flexibility - the system cannot respond to changing needs
      Workaround - Mode changes.
      Cost - More complex code and more of it; higher design
             costs.

    Loss of efficiency - caused by routines not fitting together
                         into frames of the right size.
      Workaround - slicing up several of the routines to more
                   convenient sized chunks.
      Cost - Design time, less efficient code, and loss of read-
             ability.

    Jitter - Periodic routines are placed after routines of
             variable execution length.
      Workaround - Move the affected periodic routines to
                   different frames.
      Cost - Non-optimal cyclic tuning with potential timing
             problems in other areas.
```

Program readability can be very poor in systems where several of the compromises listed above have been used. Contributing to the lack of readability is the lack of any formal method for documenting the system tuning process. The tuning process often becomes a major design activity, but in most systems nothing is said about it except for listing the calling sequences that were arrived at.

Even with complete documentation, cyclic systems tend to be very delicate beasts. The slightest change can result in timing problems in areas totally unrelated to the change, which may not show up for months, or even years.

Table 5.2. Problems Caused by Cyclic Executives.

CHAPTER 6

POSSIBLE APPROACHES

In this chapter we turn our attention to some possible ways of approaching the problems mentioned up to this point using Ada. Before proceeding to specific proposals to solve these problems, we will discuss some issues that must be addressed by any proposal that is to solve the problems presented.

The problems discussed to this point have been separated into two categories, the problems addressed by the use of cyclic executives, and the problems inherent in the cyclic executive approach.

The cyclic executive concept is used to provide a method of scheduling the processing in a system so that it will meet several types of timing requirements. These requirements include: periodic execution, maximum jitter, maximum time between executions, and maximum execution time. In order for a proposed method to be successful it will have to provide a means of imposing system timing requirements on system scheduling, provide a method for predicting system performance as compared with system timing requirements, and a set of mechanisms through which system timing can be adjusted to meet system timing requirements. Other issues that must be addressed are overhead , load leveling, and overload detection.

Many real time applications involve writing software to be run on very small processors. Scheduling algorithms that present no problem on other machines may well take too much time to execute on this type of machine.

One of the tendencies in a cyclic executive system is to move much of the event driven processing into specific time slots in the cycle. This limits the impact of a surge in events in terms of processing time. Some other method of load leveling should be available in proposals which do not provide precise time line control.

In a cyclic executive, overloads are detected very quickly (frame overrun) and appropriate action can be taken to limit the impact of the overload on the rest of the system. In systems where frames do not exist some other mechanism will have to be found for overload detection and handling.

Many of the problems caused by the cyclic executive concept stem from the process of forcing software into a single time line. While the software is being forced into this mold, it undergoes changes which alter its ability to be understood. Additionally the reasoning that underlies this process is not generally documented or otherwise maintained. In order to address these problems a proposal will either have to eliminate the frozen time line concept or move the process of placing software in the frozen time line to a level where it does not have an impact on source code and where the reasoning is preserved.

Documentation of both timing requirements and the actions taken to meet them is another serious problem with the cyclic executive approach.

Fragility of cyclic executive systems is another area in need of improvement. Each change to the software in the system could set off a chain of timing problems.

Finally each proposal must be evaluated on the basis of how generally it might be applied to problems of the type discussed. An approach that can only be applied in one or two cases is not of much use.

6.1 CODING A CYCLIC EXECUTIVE IN ADA

The first proposal we will consider involves coding a cyclic executive in Ada. This approach was proposed as a temporary measure in [26] and later reiterated in the Boeing AIMS Phase 1 report [6].

By default, coding a cyclic executive in Ada does not solve any of the problems inherent in using cyclic executives. The analysis of this approach must concentrate on how well it addresses the problems normally addressed by cyclic executives. At first, it might seem that the system designer's ability to input timing requirements into the system scheduling process would be the same as the traditional cyclic executive approach. Unfortunately, there are several reasons this is not true. First, Ada as a language was not designed with the primitive type control necessary for some types of cyclic executive. Consequently it is difficult to represent some of the primitive operations necessary for proper cyclic executive operation. This results in a clumsy implementation of the cyclic executive. Second, programs coded in Ada run on a scheduler provided by the Ada runtime system. This scheduler may vary from system to system. Therefore, a cyclic executive which works on one system might not work on another system with a different underlying scheduler. Third, even if the scheduler works on all of the various runtime systems, it is not necessarily easy to predict what a scheduler running on top of a scheduler will do.

We now turn to how well performance can be estimated in a cyclic executive system coded in Ada. Again the cyclic executive in Ada does not match up to the traditional cyclic executive approach. Since Ada is farther away from the machine than assembly language and many of the other languages used in normal cyclic executives it becomes more difficult to estimate the time it will take to execute a given piece of code. This makes the predicted performance estimates less accurate and less useful.

In the area of manipulating code to achieve timing requirements, a cyclic executive system can be manipulated with the same ideas as are normally applied to cyclic executives, with approximately the same results.

The other issues involved in solving the problems addressed by cyclic executives (overhead, load leveling, and overload detection) are handled just as they would be in any other cyclic executive system.

In summary, coding a cyclic executive in Ada does not solve any new problems, and is not as efficient in solving many of the problems traditionally addressed by cyclic executives. The benefit of this proposal over the traditional approach is questionable.

An alternative to coding a cyclic executive in Ada is to code the cyclic executive in assembly code and have it schedule Ada tasks. This approach still only addresses the problems originally addressed by the cyclic executive concept and not the problems caused by that approach.

Implementation of a cyclic executive is certainly easier in assembly language than in Ada, but the transparency of the implementation would certainly suffer from this approach. This approach also does not make clear how the assembly language scheduler would interact with the Ada runtime system scheduler. That interface would not only be crucial, but it could potentially be very complex.

Performance estimation is not improved by this suggestion since the code being analysed is still in Ada.

On the whole this proposal has the potential to be better than the previous suggestion, but its relative merit when compared to traditional cyclic executive approaches is questionable.

6.2 PRAGMATIC IMPLEMENTATION OF CYCLIC SCHEDULING

Another approach is to provide a set of pragmas to instruct the compiler to generate a cyclic scheduler. Phillips and Stevenson [35] propose a set of two pragmas: one to indicate a cyclic scheduler should be generated and another to specify for each periodic task what its period should be. This only takes into account one of the many timing constraints. This approach might benefit from some additional pragmas to input the necessary additional timing requirements and to give some information on how tasks should be assigned to frames.

POSSIBLE APPROACHES

This approach, unlike the others we have evaluated so far, not only addresses the problems ordinarily addressed by the traditional cyclic executive approach but also provides documentation of the timing requirements that the system must satisfy. This pragmatic approach, however, does not provide a complete set of documentation on all of the work done in the timing process, so the problem is only partially solved. The other problems inherent in the cyclic executive approach are not addressed at all.

With the proper set of pragmas this approach could provide the mechanism necessary to provide the needed timing and scheduling input into the scheduler design. The use of such pragmas would also help to document the system's timing requirements. The problem is that there is no guarantee that the compiler will be able to make assignments that will meet the requirements specified. In fact, since frame assignment in the real world tends to be an heuristic process with guidelines but no real rules, it would seem that a compiler that could accomplish this feat would be a very complex piece of software. A more likely solution is that the compiler would have to indicate to the user that the requirements could not be met with the existing piece of software, leaving it up to the user to figure a way to make the timing requirements work.

Analysis of timing performance is again harder for the object code generated with Ada than with assembly language or other languages used in cyclic executive systems. This drawback is counterbalanced by the power of the Ada language.

The mechanisms necessary to adjust timing in this approach are the same as used in the traditional cyclic executive approach.

This proposal presents an attractive way to input timing requirements into a cyclic system. If the complexity problems of the compiler could be solved this proposal would provide a method of generating a cyclic executive competitive with the traditional approach in some cases. This proposal does not have any pretensions of solving any of the problems normally associated with cyclic executive systems. The only attempt at improvement is the partial documentation of timing information which is counterbalanced by the difficulty of analysing the timing properties of Ada code.

6.3 DELAY STATEMENT SCHEDULING

The delay statement can be used in many ways to impact the scheduling of processing. In the approach to be evaluated here the delay statement will be used to schedule start times for every periodic task.

This approach is ambiguous as to how scheduling is supposed to work. It is possible to try to define a scheduling time line such that only one task will be enabled at a time, thus simulating a cyclic executive. It is also possible to say the delay statements just provide start times. Once tasks are started they are meant to compete for system resources. Enabling one task at a time just provides a less precise method of specifying the same frozen time line as a cyclic executive, and so creates all the disadvantages of a cyclic executive without the benefits of the precise specification of what happens when frames overrun etc. To evaluate this proposal we will concentrate on the second idea.

In this approach only start times and periodicity can be specified, leaving the other timing concepts up to the designer. Consequently jitter requirements, maximum time between executions, and maximum execution times depend on the design methodology. This means the input to scheduling is only partially addressed by this approach.

Performance prediction is also left up to the designer and is somewhat more complex than for a cyclic executive. Because a more complex scheduling mechanism controls several competing tasks. In order for performance prediction to be done at all, several assumptions must be made about the run time system and its scheduler. These assumptions are similar to the assumptions detailed under the next proposal. They involve knowing every piece of execution time used in the system, exactly what the scheduler will do, and when it will do it. With this knowledge the system designer can walk through the timewise execution of his software, trying to anticipate trouble spots. Depending on the complexity of the system, this approach can be very difficult.

In terms of mechanisms to adjust the timing of the system to meet the requirements, they certainly exist, but just what mechanisms to use for each timing problem and their side effects are not very clear. A likely way to adjust timing would be to move the start times of interfering tasks farther apart, but how effective this is and how far apart they have to be moved depends on what other tasks might impinge on the new start times (i.e. moving a task's start time into the run time of a higher priority task is not likely to help).

This proposal does directly address the cause of most of the problems with cyclic executives. The software in this proposal is not forced into a frozen time line. Therefore the potential exists to eliminate the evils inherent in the source code manipulation required to make such a time line work. Given the unknowns of this proposal, however, it is difficult to assess how completely these problems will be solved.

A suggestion to improve this proposal would be to provide the capability to delay until a specified time. Each periodic task is really computing the time it wants to run next. Converting time into a delay not only seems to be less efficient, but also provides a less accurate scheduling of the next run time. Specifying a delay interval means that two routines that are supposed to become ready to run at the same time, might not be ready to run at exactly the same time due to small delays between the time the delay is computed and the time the scheduler works on the delay request.

In summary, this proposal relies heavily on the system development methodology used to achieve its timing requirements. With the proper specification, design, analysis, and manipulation methodologies this could be a very effective technique for real time design. Without them it does not really provide much capability.

6.4 DATA DRIVEN DESIGN TECHNIQUE

The Data Driven Design technique involves having input and output tasks driven periodically either by external events or delay statements and having the rest of the system data driven. This approach gives the system the ability to adapt to changing data requirements and yet retain its periodic input and output requirements. The periodicity of data entering and leaving the system will force periodic execution on the processes dependent on that data. An example of this type of system is shown in Figure 6.1.

Again this approach relies heavily on design methodology to achieve its timing requirements. The only timing considerations that can be input as part of the design are the periodic constraints on the input and output routines. Even the periodic requirements of these routines must be verified through the use of timing analysis methodology. Jitter requirements, maximum time between executions, and maximum execution times are all left completely to methodology. One additional timing requirement which is extremely critical in this type of system is the time from input to the time an output is generated. This is especially important in feedback systems.

This type of system lends itself to an SADT* type methodology very well. If SADT were extended to include all the timing constraints needed it would provide an excellent framework for timing specification and analysis within the system.

Before timing analysis can be performed on this type of system several assumptions have to be made about information available to the system analysis personnel. First the scheduling algorithm used by the Ada implementation must be known. The best type of scheduler for this type of system is a preemptive scheduler. Each time a change can be made in the ready-to-run table, the scheduler is called and the highest priority task is run. If all tasks are of the same priority the task that has been ready to run the longest is selected. Additionally, the scheduler run time must be known (most likely as a function of the number of tasks and the number ready to run). The run time of each program segment or language construct must be known (this would be a convenient thing for the compiler to supply). Finally, either care must be

----

*SADT is a trademark of SofTech Inc.

taken in coding so that garbage collection does not occur, or the schedule and run time of the garbage collection algorithm must be known.

Once the information above is known, timing analysis can take place by following each data path from input to output propagating minimum, maximum, and average times. Jitter problems will show up as higher priority tasks that can potentially impinge on period sensitive routines or as variable length data paths. Maximum time between execution violations will show up as high priority routines that can combine to execute for an extended period of time. Maximum execution times will fall directly out of the algorithm. When timing problems are found they can often be addressed through changes in data flow (such as new buffering schemes) or by a change in priority.

The resulting timing analysis system will be extremely complex. It is unlikely that any meaningful timing analysis could be performed without the asistance of a timing analysis tool. The result of the analysis will also be complex potentially requiring the assistance of a software tool to assist in the interpretation of its meaning.

This proposal does not require software to be fitted into a fixed time line, relieving the designer of most of the worries associated with cyclic executives. All timing control flows from the input and output tasks leaving the source code relatively intact.

There are only two problems with this approach. First, the timing analysis is quite complex. It may be difficult to recognize potential timing problems, and if they are recognized, it may be difficult to estimate their severity. Second, it is possible that this type of system could inherently have more timing problems because its time line is not fixed and it has more overhead. Processing overlap between routines is only estimated, eliminated. Cyclic executives explicitly rule out processing overlaps between critical routines.

A final area of worry with this proposal is how generally it can be applied to routines that would previously have been approached with a cyclic executive. This is especially worrisome in a system with a very small processor where the additional overhead of this approach might cause failure, or the unfixed time line might cause spot overloading to degrade the system in some unanticipated fashion.

In summary, when the proper methodologies are followed this provides an excellent method for programming most real time systems. Its only major problem involves the complexity and ultimate reliability of the timing analysis methods. Perhaps a tool to aid in this activity would be worth investigating.

6.5 A TRANSITIONAL APPROACH

The final proposal involves taking a transitional approach to the problem. Source code would be developed as in the last proposal but additionally all timing requirements would be developed as well. After program compilation the resulting code would be restructured into a cyclic executive using an interactive knowledge based scheduling tool. This type of approach would combine the development and maintenance advantages of the data driven approach with the run time advantages of a cyclic executive.

A timing requirements definition phase would be necessary to provide a method for the specification of timing requirements in such a system.

This type of system directly answers both the problems addressed by cyclic executives and the problems caused by cyclic executives. All timing requirements are presented to the system as a separate input from the source code, providing the system with knowledge of both but not confusing one with the other. Timing analysis would have to be facilitated by the scheduling tool as part of developing the cyclic executive. The mechanisms for manipulating the code to fit the scheduling pattern would also have to be part of the tool.

This approach suffers from one major drawback. During system integration with the target hardware, the integration personnel are tempted to patch the compiled code rather than modify the source and recompile. This temptation would be further exacerbated by the need to run the compiled code through another step in order to have it scheduled in a cyclic executive form. This method could have two bad effects. First, the timing of the cyclic system could be upset by the patched code, causing all sorts of problems to appear that really are just timing errors. Second, when the patch has to be incorporated into the source code, the translation between the patch and the source update is likely to be nontrivial.

In summary, this proposal has a great deal of potential although there are some problems to overcome.

6.6 SUMMARY

The proposals that have been presented can be classified in three groups. The first three proposals try to transplant the cyclic structure directly into Ada. They then have the common problem of not being able to rise above the problems inherent in the cyclic structure. These problems are significant, especially considering these are some of the problems Ada was introduced to solve in the first place.

The second group of proposals try to use the power inherent in the Ada language to solve the same class of problems as the cyclic executive. These suffer from a heavy reliance on development methodology in order to provide the necessary performance. The analysis involved in this methodology is both time consuming and complex, but it could be facilitated through the use of a few simple tools.

The transitional approach, although very powerful in terms of dealing with the problems we have been addressing, requires an extremely complex tool in order to function and introduces a new set of problems of its own.

The ranking of the proposals discussed versus the stated requirements is shown in Table 6.1.

The first set of proposals have the dual unfortunate aspects of defeating the purpose of Ada and of not being any more effective than the traditional cyclic executive approach. These should not be pursued any further. The transitional approach requires a tremendous amount of further investigation which might be a good topic for another study. The data driven design approach also deserves further study. With the appropriate methodologies and a few simple tools, this approach could be useful in the design of some real time systems.

Proposal	Periodic Timing Requirement	Maximum Jitter Requirement	Maximum Time Between Executions	Maximum Execution Requirement
Cyclic in Ada	4	3	4	3
Cyclic in Assembly	4	4	4	3
Cyclic Pragma	5	5	5	5
Delay Statement	4	1	2	2
Data Driven	4	1	2	2
Transitional Tool	5	5	5	5

5 = very good, 1 = very poor

Table 6.1a. Proposal Ratings, Designers Capability to Impact Scheduling

Table 6.1a Explanation of Categories

[1] Periodic Timing Requirements - This indicates how easily a system designer can direct a piece of software to have periodic timing characteristics.

[2] Maximum Jitter Requirements - This indicates how easily a system designer can direct a piece of software not to exceed a specified maximum jitter from its periodic rate.

[3] Maximum time between execution - This indicates how easily a system designer can direct a piece of software not to have more than a specified time between executions.

[4] Maximum Execution Time - This indicates how easily a system designer can direct a piece of software not to exceed a specified maximum execution time.

proposal	Load Leveling	Overload Detection	Overhead	Predictability of Performance
Cyclic in Ada	5	5	4	3
Cyclic in Assembly	5	5	5	3
Cyclic Pragma	4	4	5	3
Delay Statement	2	2	2	3
Data Driven	2	1	2	2
Transitional Tool	5	4	5	5

5 = very good, 1 = very poor

Table 6.1b. Proposal Ratings, Performance Issues

Table 6.1b Explanation of Categories

[1] Load Leveling - Indicates the proposal's ability to schedule processing from event driven requirements in such a way that they do not impact the rest of the system during a burst of events.

[2] Overload Detection - Indicates how well and how early the proposed scheduling mechanism can detect an overload situation so that some action can be taken.

[3] Overhead - Indicates how much processing time the scheduling algorithm might take, as compared to total available processor time in some small systems.

[4] Predictability of Performance - Indicates how easily the user of this proposal will be able to estimate the timing performance of his software.

Proposal	Non-Segmented Source Code	Timing Requirement	Tuning Documentation	Scheduler Readability
Cyclic in Ada	1	1	1	5
Cyclic in Assembly	1	1	1	4
Cyclic Pragma	1	5	1	3
Delay Statement	4	2	3	2
Data Driven	5	1	3	1
Transitional Tool	5	5	5	4

5 = very good, 1 = very poor

Table 6.1c. Proposal Ratings, Readability

Table 6.1c Explanation of Categories

[1] Non-Segmented Source Code - Indicates how little the source code has to be modified in order to meet timing requirements.

[2] Timing Requirement - Indicates how well documented the timing requirements are if the system is developed using this proposal.

[3] Tuning Documentation - Indicates how well the actions necessary to tune the system are documented.

[4] Scheduler Readability - Indicates how easy it is for the user to perceive the actions the scheduler is going to take.

Proposal	Fragility of Algorithm	Generality of approach	Ease of implementation	New Problems introduced
Cyclic in Ada	1	5	5	5
Cyclic in Assembly	1	5	4	4
Cyclic Pragma	1	5	4	3
Delay Statement	4	4	3	4
Data Driven	5	4	3	4
Transitional Tool	3	5	1	1

5 = very good, 1 = very poor

Table 6.1d. Proposal Ratings, Other

Table 6.1d Explanation of Categories

[1] Fragility of Algorithm - Indicates how likely coding changes are to cause timing problems.

[2] Generality of Approach - Indicates how many of the problems approached traditionally by cyclic executives can be approached using this proposal.

[3] Ease of Implementation - Indicates how involved the development of the necessary methodologies and software will be for this proposal.

[4] New Problems Introduced - Indicates this proposal's propensity for introducing new problems.

CHAPTER 7

EFFICIENCY ISSUES

This chapter examines the role of the optimizing transformations for the Data Driven Ada programs for real time systems. Data driven designs offer significant advantages over traditional real time designs in terms of understandability, documentation, and maintenance. The data driven design has, however, two major drawbacks: significant tasking overhead in current Ada implementations, and a lack of determinism in system timing performance. These drawbacks are serious but do not, of themselves, preclude the use of data driven designs in real time systems. This chapter examines the use of optimizing transformations to improve these two aspects of a data driven design in critical areas. The motivation behind the use of this sort of transformation is to improve the performance of a system by the use of strategic optimizations until adequate performance has been achieved. These are not suggested coding styles. These are transformations, to be applied after coding is complete and timing problems have been identified. This chapter evaluates some of the areas in which existing implementations of Ada might benefit from optimization and surveys some of the most effective existing optimizations, and suggests possible areas for further optimizations.

7.1 MULTITASKING AND SYNCHRONIZATION

One of the major design goals for the Ada language was to reduce the life-cycle cost for real time embedded software. The language achieves this goal by supporting modern software engineering methods as well as many real time concepts. Ada does not, however, guarantee any reduction in life-cycle costs if real time designers and programmers do not take advantage of the potentials of the language. Developing software with Ada according to traditional real time design practices (such as cyclic executives) is unlikely to have a significant cost impact during the program life-cycle. In order to achieve significant reduction in the life-cycle cost of real time software, it will be necessary to design software in a fashion which takes advantage of the strengths of the Ada language.

The data driven methodology proposed is designed to utilize the strengths of Ada during the development of real time software. This methodology, however, does not guarantee adequate software performance in all cases. In this chapter we address the idea of transformations to improve the performance of time critical code. These transformations improve performance either by producing a new source module with less overhead, producing a new source module that behaves more deterministically, or utilizing a special run time feature (such as a fast interrupt handler) to improve performance. We will study two classes of transformations, both of which decrease overhead and increase determinism: those which reduce the number of tasks in the system,and those which reduce the number of synchronization points.

7.1.1 The Need For Tasks

In a real time system a large number of tasks are often introduced for several reasons. The most important reason is that conceptually, the many repetitive activities of the system can best be modeled as tasks with independent threads of control. Furthermore multiprocessing is seen as a way of improving the throughput of the system. If a separate task is delegated to handle each data item in a queue, then a throughput improvement may be possible if each of these tasks needs to synchronize with other activities in the system. In Ada, however, there is another situation which leads to proliferation of tasks: programming of certain synchronization disciplines (e.g. monitors, agents, etc.) requires additional tasks for protection of data.

7.1.2 The Need For Synchronization

The most important need for synchronization arises when a data transfer involves various independent threads of control or when there is a need to access shared data in a potentially destructive manner. In addition, there are implicit events (not directly evident from the program) such as process creation, destruction, shared variables, and rendezvous which will depend on some form of synchronization. Furthermore, in the presence of multitasking, the implementation of exception handling functions may also entail some implicit synchronizations.

7.2 OVERHEAD OF MULTIPROCESSING AND SYNCHRONIZATIONS

Proliferation of tasks and synchronization points introduce certain overhead in the real-time systems designed for single processor use. First, each task needs to be represented at runtime. Additional memory space is necessary for task attributes such as its stack, stack frame, task body, task object, etc. Therefore, the space available to the program is reduced and extensive memory management is required at runtime.

In Ada many simple synchronization needs such as semaphores, monitors, agents, etc. require the use of additional tasks which may not contribute to the throughput of the system. In fact the presence of these tasks may hinder the achievement of appropriate throughput.

Most importantly, a proliferation of synchronization points, such as those discussed above, may also lead to a proliferation of scheduling points*. This is especially true of the class of pre-emptive schedulers (where there must be a scheduling point every time a task is unblocked).

The need for implicit synchronization also arises from the subtle semantic ramifications of the Ada Language definition, such as: storage allocation, task creation, task termination, entry calls, accept statements, delay and terminate alternatives, and abort statements to name a few. The exact overhead and language requirements should be understood for the discussion of any possible optimizations which could result.

[1] Entry Calls: An entry call must synchronize with the corresponding accept statement. Furthermore, a check must be performed at the time of entry call to determine if the calling task has become abnormal. If the calling task has become abnormal it must be completed.

[2] Accept Statements: In addition to the required wait for an entry call, the accept statement must check for the abortion or termination of the calling task as well as for a timed entry call. According to Ada semantics, an entry call is cancelled if the calling task is abnormally terminated and the rendezvous has not started. Also if the accept statement is reached before the time-out of the conditional entry call, the timer must be reset. Furthermore, if the called task is abnormal, all entry calls are cancelled and the task becomes completed. Even though this overhead may not seem too drastic when considered for individual cases, together they can noticeably affect the efficiency of task synchronization.

[3] Abort Statements: When a task is aborted, it becomes abnormal. However, it is not required to become completed immediately. All

* A scheduling point is one where a potential context switch may be necessary, if the synchronization partners of a given task are not ready.

further interactions with the aborted tasks are forbidden (in particular rendezvous). There could be two possible implementations of an abort statement, and each possibility has different implications. We shall call the first one synchronous abort and the second one asynchronous abort. A synchronous abort completes when the aborted task is completed. This method ensures that all dependent tasks are completed and terminated as well, but it could take an indefinite amount of time. In an asynchronous abort, the aborted task is marked as abnormal and the execution of the abort statement is over. The abnormal task becomes completed at the next synchronization point. This approach has the advantage that no waits are involved, but again it is not known when the task and its dependents are to be terminated. Each possibility requires a potentially massive amount of synchronization.

[4] Task Creation: In Ada, task creation is done in two logical phases: the act of creating a task object and that of activating a task. It is the latter which requires some degree of synchronization with the parent unit, and some checks need to be performed. First of all, the start and end of a task activation have to be synchronized with the parent unit or with the unit which causes the task activation. Furthermore, due to the semantics of the abort statement, a check must be performed to see if the task has become abnormal. This check must be perfomed at various levels: The task activator must check if it is abnormal and become completed if so. Similarly a check must be made for the task which is being activated before activation and when the activation is complete. A similar check may be neccessary for the activator to determine if an abort has been issued for it.

[5] Task Termination: The act of task termination, in general, requires some form of synchronization with certain other tasks. In simple cases a task termination must coincide with the termination of all dependent tasks.

[6] Storage Allocation: The strategy chosen for dynamic storage allocation for the class of system and user-defined objects may introduce synchronization points leading to "invisible overhead". For example, if a central heap is chosen by the implementor for handling all storage allocation requests, then it may introduce contention among all tasks which need storage. It requires that the heap or the storage manager be encapsulated within a conceptual monitor task. This forces implicit synchronization among a set of seemingly independent tasks, leading to unpredictable waiting times, unnecessary flow control, and complicated algorithms for allocating and deallocating storage.

7.3 POSSIBLE OPTIMIZATIONS

It is evident from the foregoing discussions that two different classes of optimizations are needed. Implementation independent and implementation dependent. Implementation independent optimizations are intended to remove overhead that arises due to certain stylized manners in which tasking constructs are used. However, very little is known at this time regarding the tasking styles that are likely to be used. Even for some of these styles, source to source transformations are not possible. Thus the programmer is forced to depend on specialized support for such idioms, other than implementing his own version of an appropriate tasking model in sequential Ada.

7.3.1 Implementation Independent Optimizations

Currently there are very few implementation independent transformations known for concurrent programs in general. Moreover in Ada, these are even rarer because Ada does not offer many concurrent programming constructs that are simpler than entries and tasks. Thus we only refer to one optimization we know: Program Inversion, which is fully described in [38]. The general idea is to transform a set of tasks that transform a stream of data into a single program requiring no concurrency at all. In the general model where this applies, each task operates on a single datum of the stream, transforms it according to some requirements, and passes the transformed datum to the next task in the sequence. In the inverted program, the entire system of tasks is replaced by an infinite loop that consists of the sequences of transformations, performed by the individual tasks. Thus several tasks are replaced by a single task which runs the original task components in a specified order. This lowers tasking overhead and creates a very deterministic system.

7.3.2 Implementation Dependent Optimizations

There are a variety of implementation dependent optimizations which are possible, and they have been widely reported in the literature. The best known is the so-called Habermann-Nassi Optimization [20], which has spawned a number of similar strategies [17,24,37]. We shall briefly review them here and then address the general issues.

The Habermann-Nassi optimization was one of the first optimizations designed to minimize the number of context switches that occur in the course of a rendezvous. Depending on the task that arrives at the rendezvous point first, two or three context switches are required because the accept statement is normally executed by the called task. What is required, however, is that the call be executed in the context of the called task. This class of optimizations, retains the synchronization point but performs the execution of the entry call based on task the arrival order. This method requires some transformation of the entry call and accept statements to some sequences of calls to the runtime library routines. By their very nature these must be performed by the compiler.

Hilfinger considers a class of idioms known as monitor clusters [24]. These are a group of monitorlike tasks that only interact amoung themselves. The tasking and synchronization implementations within the monitor cluster can be greatly simplified when compared with typical tasking and synchronization implementations. This simplified implementation removes a great deal of the overhead discussed above. This simplification strategy can be extended to also deal with agent tasks. Through this method whole groups of tasks can be identified that do not require a full tasking implementation greatly reducing the average overhead of a tasking operation.

Rajeev [37] considers several specialized strategies to reduce tasking overhead in specialized cases. He is primarily concerned with an initial rendezvous optimization, but several more general optimizations form components of this optimization. He shows that an initial rendezvous can be reduced to a simpler deterministic sequence of operations. In the process he shows several more optimizations. First, when an entry only receives calls from only one task the entry call queueing mechanism for that entry can be eliminated. Second, in situations where the order of an entry call and entry accept can be predetermined, synchronization can be eliminated.

Greene [17] has implemented a special class of rendezvous for handling interrupts. The basic idea in this scheme is to treat the implementation of interrupt rendezvous as similar to that of a procedure call. In this approach, the task switch is performed with minimal state storing operations. The underlying assumption, however, is that there are no control transfers or synchronizations performed from within the context of the rendezvous. The compiler depends on the user to indicate when such transformation is required through the use of special pragmas. Of all the implementation dependent optimizations, this is the only implemented one we know of. The resulting performance is impressive.

7.3.3 Other Possible Optimizations

Section 7.2 lists many causes of overhead in current Ada implementations. Each of these needs to be evaluated in terms of whether or not it can be avoided in a significant class of cases and if these cases can be easily identified, will the elimination of the overhead result in a significant time savings. One example is where a program contains no abort statements, it might be possible to eliminate much of the code required to check for abnormal tasks. Possible other ideas for this type of optimization include:

- o When there are no timed entry calls to an entry it need not incorporate the processing to handle them.

- o There can be several options in terms of storage allocation. These might include: private heaps, predefined pools of objects, or static allocation of some storage.

7.4 SUMMARY

In this chapter we have barely scratched the surface of the possible optimizations. We have, however, pointed out many of the areas where overhead arises in straightforward tasking solutions to common problems. In order to develop an effective set of optimizations, the first step is an exhaustive analysis of the various time consuming steps in tasking operations. This analysis can show how: code can be transformed so it will require less overhead, specialized pragmas such as in [17] can be used to provide less overhead and more deterministic behavior, and code can be transformed to behave in a more deterministic fashion.

CHAPTER 8

TEMPORAL MODELS OF REAL TIME SYSTEMS

It has been recognized that any approach to designing a real time system must include a method of evaluating the temporal behavior of the software produced in order to verify that timing requirements will be met. The traditional approach to the design of real time systems is to use a cyclic executive. This provides a simple timing model upon which system performance estimates can be based. This approach has two drawbacks: it leads to high life cycle costs, and it is not adequate to express the requirements of many of the more complex systems being developed today. A great deal of work has been done in trying to build timing models that can represent a more general class of software. There is, however, no general solution to the problem available today

This chapter will first discuss what requirements must be addressed for a method of temporal behavior analysis to be generally applicable for real time system development. These requirements will then be used as a measure in the discussion of the timing analysis techniques that have been developed. Finally, there will be a discussion on how some of these techniques might be combined heuristically in order to form an analysis technique capable of meeting the needs of today's real time community.

8.1 REQUIREMENTS FOR TEMPORAL BEHAVIOR ANALYSIS

To understand the requirements for a performance estimation system in a real time environment, we must first understand the role of performance estimation in this environment.

In a real time system it is as important for the software to meet the system timing constraints as it is for the software to perform the right function.

Consequently, the issue of timing performance must be addressed in some form at every level of design. Each design decision must be evaluated not only in terms of its effect on the system function, but also in terms of its effect on system timing. This analysis is accomplished through some means of performance estimation. At each level of design, the performance is estimated for the lowest level elements of the system (as defined at that stage in the design). These timing estimates are combined to form a model of the overall system. This model is then examined to determine if timing performance falls within the required bounds. This model can also be used to evaluate what effect various design decisions might have on system timing.

Timing performance estimation should be available from the earliest stages of system design. As the design is decomposed, the timing model of the resulting system is also decomposed. At each step in the design, the timing model is analyzed. This gives the designer the opportunity to develop a strategy for creating software that will be able to meet timing requirements. This also uncovers many serious timing errors early in the design cycle, where they are far less costly to correct. The purpose of this timing model is to allow the designer to evaluate the timing implications of the various design decisions to be made. Manipulations of the system software solely for the purpose of meeting a specific timing requirement (as opposed to a design decision that has to be made anyway, and in which timing is considered) should not be performed until the system is completely coded.

The required type of performance determination changes over the course of system development. Early in system development it is difficult to define exact characterizations of the performance of the design elements. At this stage in the design, it is highly advantageous to use statistical performance models of the system. This method also appeals to common sense because it would be premature to make absolute statements about the performance of the system. As the design becomes more concrete, the information available regarding the timing performance of the system becomes more accurate. At this point, it becomes reasonable to start assigning more exact formulations to the run time for each design component and arriving at more concrete measurements of system performance. When the system is coded, it becomes possible to arrive at exact formulas for the description of the execution time for any piece of sequential code. These descriptions can be combined to form a very precise model of the system. This model can be used to make concrete statements about system performance.

The ideal timing performance estimation technique must support a wide variety of reasoning based on its underlying model. The simplest form of this reasoning is verification that timing requirements will be met. The model should also be able to produce numeric estimates of the actual system performance as compared to required performance. The final and most difficult type of analysis to be supported is the determination of where, within the design, the cause of a particular timing problem is found. The difficulty here lies in determining the effects of unrelated parts of the system on the execution of the relevant portion of the system. The first step in attacking this problem is the decomposition of the system timing constraints into constraints on system components. These decomposed constraints can then be compared to the performance of the components. This comparison should point to some spots in the design that might be contributing to the observed problem.

Conventional software development techniques have abstracted away from the timing properties of the system. This is especially true during early design stages. While this has produced many advantages for the conventional software designer, it is clearly inadequate for the real time designer. On the other hand, traditional real time design methods neither enforce enough separation between timing and functional considerations during system design nor take advantage of many of the innovations found in modern software engineering. This failure results in expensive, fragile, unmaintainable, and unreliable systems. The ideal design technique must allow interaction between timing and functional considerations, but those considerations must be kept separate for as long as possible. Additionally, the interaction must be confined to a well defined interface.

There are many timing measures that must be obtained depending on the timing requirements of the system being developed. In some cases these may be statistical in nature, but in other cases the measures must be deterministic quantities. Some measures that are required follow:

- Maximum Response Time
- Average Response Time
- Average Jitter (Periodic Routine)
- Maximum Jitter (Periodic Routine)
- Throughput
- Processor Utilization

Various combinations of these requirements may be present in any given real time system. The same part of a system may even be covered by more than one type of requirement (i.e. the same component may have both a response and a throughput requirement).

Figure 8.1 summarizes the requirements for temporal analysis in real time systems.

8.2 METHODS OF TEMPORAL BEHAVIOR ANALYSIS

We now turn from what is required of temporal behavior analysis to discuss some of the methods that have been proposed to meet these requirements. This section will contain a brief introduction to several of the techniques that have been proposed. The next sections will take a deeper look at some of these techniques.

Unfortunately, no single method of performance estimation can meet all the requirements for an ideal method of performance estimation. At best a single method of performance estimation can determine some timing properties of some types of systems.

Many methods have been proposed for determining the timing behavior of programs. Some can be used easily throughout various design stages, others have to be adapted to fit some stages of the design process.

Two criteria must be met for each of the methods for determining timing behavior: first, a method of modeling the software system; second, a method of evaluating the model for the desired performance estimate. First, we will examine several methods of modeling software performance and then we will discuss ways of evaluating these models.

Each method of modeling a software system has its good and bad points. Some of the issues for determining the correct modeling technique are:

- Whether statistical or deterministic time consumption is represented.
- Level of detail that can be represented.
- The power of the modeling technique to represent a variety of systems.
- The complexity of preparing the model to be used.
- The complexity of analyzing the model once generated.

1) Temporal analysis must be available at every step in the design process from inception to code.

2) The analysis technique must be capable of expressing timing in either statistical or deterministic terms, or possibly some mix of the two.

3) The analysis technique must be able to find answers for a variety of questions, including:

- Does the system meet a particular constraint?
- What is the measurement of system performance versus that constraint?
- What parts of the system influence that measurement?

4) The analysis technique must support a variety of timing measurements, including:

- Maximum Response Time
- Average Response Time
- Average Jitter
- Maximum Jitter
- Throughput
- Processor Utilization

Figure 8.1. Requirements For Temporal Analysis

Many models have been used to represent software performance. Some representative examples include:

- o Queueing Models
- o Timed Petri Net modeling
- o Abstract Process Networks
- o Markov Chain modeling
- o Finite State Automata
- o Computation Structures

Each of these will be described briefly in the following subsections, in terms of its major features, advantages, and disadvantages.

8.3 TEMPORAL BEHAVIOR ANALYSIS WITH QUEUEING MODELS

Queueing models are a very high level representation of system performance. A queueing model treats each system resource as a service center. Processing which is to use each service center must wait in a queue until the service center is available. This approach provides a good high level statistical description of some systems. Unfortunately, it is difficult to decompose these models into a detailed system model. This type of model is also limited in the type of system it can represent. Some systems just cannot be represented easily as a queueing model.

As with many high level modeling techniques, effective queueing models require a great deal of expertise on the part of the designer. The reason is the complexity of mapping real world problems into their very abstract representation. When so much information is abstracted out of a model, it becomes difficult to ensure all relevant information is still represented.

Queueing models represent a system in terms of system resources and a statistically characterized load. Each system resource that must be shared by the various system processes (i.e., processor, disk, printer, special hardware interface, etc.) is represented by a server queue mechanism. The various jobs that require the use of this resource wait on a queue until the resource is available. Jobs on the queue are handled on a first come first serve basis. There can be one or more types of jobs in the system. Each type of job has a characteristic service demand (how much time it takes for the resource to serve it) for each of the resources it uses. Each type of job also has a characteristic workload intensity (or number of jobs likely to request services over any given time period). These two quantities characterize the load on the system.

The primary challenge in building an accurate model is choosing the correct method of representing system workload. There are several standard methods of representing this load - a transaction workload, a batch workload, and a terminal workload. A transaction workload models the typical transaction processing system where workload is characterized by the average rate at which requests (jobs) arrive. A batch workload models the typical batch processing system where the workload is characterized by the average number of active jobs. A terminal workload models the typical time sharing system where the workload is characterized by the number of users and the average think time between user requests. While these provide adequate characterizations of many systems one of the challenges of using queueing theory to model a new system is to pick just the right type of work load representation. For many systems, it may be necessary to build a new method of workload representation. Examples of this are shown in [22] and [23].

It is clear that as the software design of the modeled system becomes more detailed, it becomes more difficult to reflect this more detailed information in the workload characterization. In fact, it would be foolish to try because the queueing network model derives its power from a very high level view of the overall hardware and software system. Adding extra detail in one area and ignoring the detail of the other areas can be dangerous. In order to gain anything from a more detailed software model, more attention would also have to be paid to system scheduling details, hardware details etc.

Because of queueing models' inherent high level approach to system performance, they provide a very powerful tool during early system design stages. Queueing models can provide valuable information on system sizing questions, load distribution, and system level bottlenecks. During more detailed levels of the system design other performance models are needed.

A more complete description of queue networks and their application to performance analysis problems can be found in [25].

8.4 TEMPORAL BEHAVIOR ANALYSIS WITH PETRI NETS

Timed Petri net modeling involves using a time extended version of Petri net theory to represent the software system. Various types of time extension have been proposed. Also, various types of Petri net models have been suggested, including Petri nets, subsets of Petri nets, and extension of Petri nets. Petri nets can be used to represent a wide variety of different systems with both statistical and deterministic descriptions of the design elements. Unextended Petri nets cannot represent the function of all software because there is no test for zero (actions can only be triggered by the presence of a token, not by the absence of any tokens), but any situation that does not call for that type of test can be represented (this drawback is not serious for representing system time performance). The most serious problem with timed Petri nets may well be that they are too flexible in the type of systems that they represent. This excess flexibility sometimes causes problems in determining the relationship between a given net and a piece of software. Building a Petri net to represent the timing behavior of a piece of software can sometimes be hampered by a single type of Petri net element representing a wide variety of different things in the software system. Situations then arise where the mapping between the software and the Petri net may become hard to comprehend. An additional problem is that the Petri net representations of systems of even moderate complexity can become so complex as to be almost unreadable.

The most familiar representation of a Petri net is the Petri net graph. Here, a Petri net is represented as a bipartite directed multigraph. This graph consists of two types of nodes (places and transitions) connected by directed arcs. Places may only be connected to transitions and transitions may only be connected to places. There may be more than one arc between the same place and transition. The state of a Petri net is shown by tokens which reside in the places of the graph. An arbitrary number of tokens can reside in any particular place. A Petri net transitions from one state to the next when a transition fires. In order for a transition to fire it must be able to get one token for every input arc from the input place attached to that arc. When the transition fires a token is removed from the place corresponding to each input arc, and a token is added to the place corresponding to each output arc. This type of graph can be used to model a wide variety of systems. Figure 8.2 shows a simple model of a missile launcher. When the system receives a load launcher command, a warhead and a missile are removed from stores, they are combined to form an operational missile and loaded on the launch rails. When a missile is loaded on the launch rails and the command to fire is received, the missile is launched. Tokens are used to model the missile, the warhead, the load command, and the fire command.

Petri nets and Petri net subsets and supersets provide a tremendous amount of modeling power. Standard Petri nets fall just short of Turing machines in their ability to model software. The one addition needed to make them equivalent is the ability to test for zero. This addition might be necessary for the modeling of the functional aspects of software but is not necessary to represent the temporal aspects discussed in this chapter. To model the temporal behavior of a software system, it is only necessary to know the paths that the software might execute, not how those paths are selected. For example, when modeling an "if" statement, it is adequate to show that one of two paths will be followed, not how the correct path will be selected.

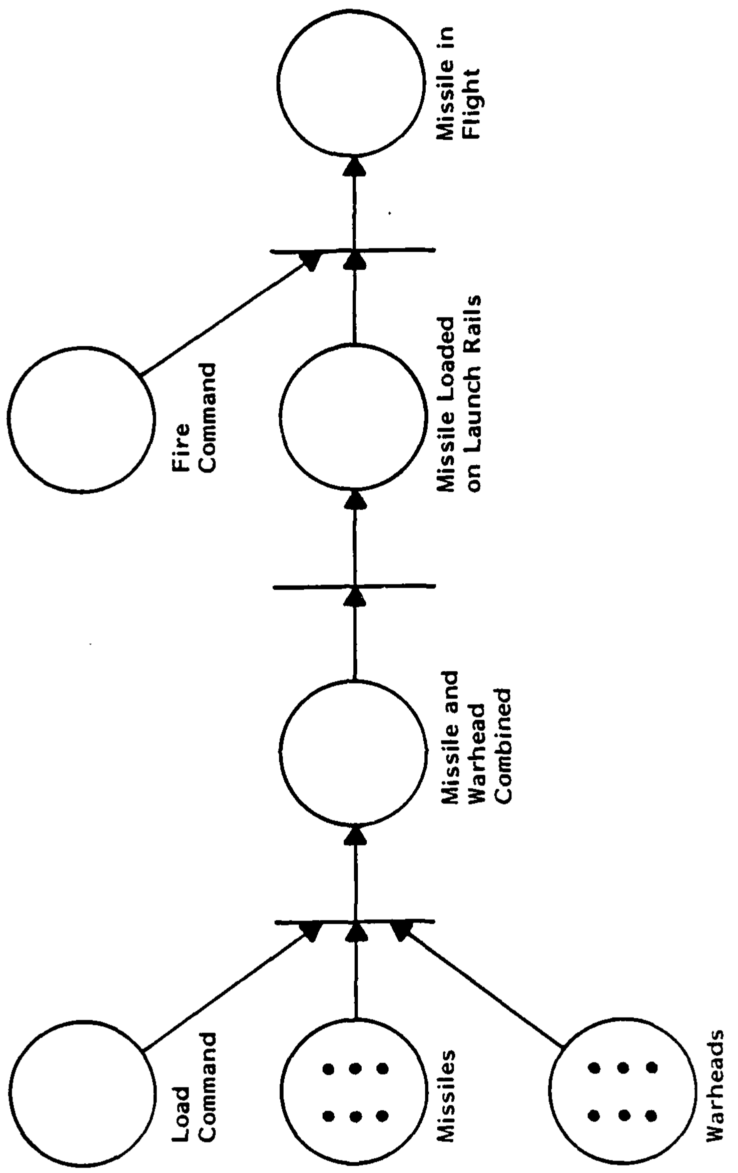


Figure 8.2 Petri Net Model of a Missile Launcher

TEMPORAL BEHAVIOR ANALYSIS WITH PETRI NETS

In order to represent the temporal behavior of software, Petri nets need to be extended to incorporate the concept of time. Time consumption within the Petri net model can be represented in several ways; [39] suggests that each transition be given a time attribute which corresponds to the time it takes for the transition to fire after being enabled. More recently in [10], it has been proposed that time consumption might be better modeled by placing a time attribute on each place corresponding to the time that a token must reside in that place before being available to an output transition. Figure 8.3 shows how an "if" statement would be represented according to this scheme. This approach has the advantage of maintaining the original Petri net definition of an instantaneous transition from one state to the next. When transitions are not instantaneous there will be times (during transition) when the model will be in no identifiable state.

The type of time delay associated with either method of representing time consumption can vary as well. These delays can be represented as deterministic quantities, or they can be represented as random variables (stochastic Petri nets). Either representation has its advantages: random variables early in the design, and deterministic quantities in the detailed stages of design and coding. If the delays in a stochastic Petri net are distributed either exponentially or geometrically then the Petri net is isomorphic to a homogeneous Markov chain. These Markov Chains are discussed in [2].

The major drawback of this scheme is that it is too general. Since we are only going to be modeling software systems there is no need to be able to model any type of system. Additionally, by limiting the modeling power the complexity of analysis can also be limited. The issues involved in creating a more software directed version of Petri nets is addressed in the next section on AP networks.

A more complete description of Petri nets and how they can be used can be found in [34].

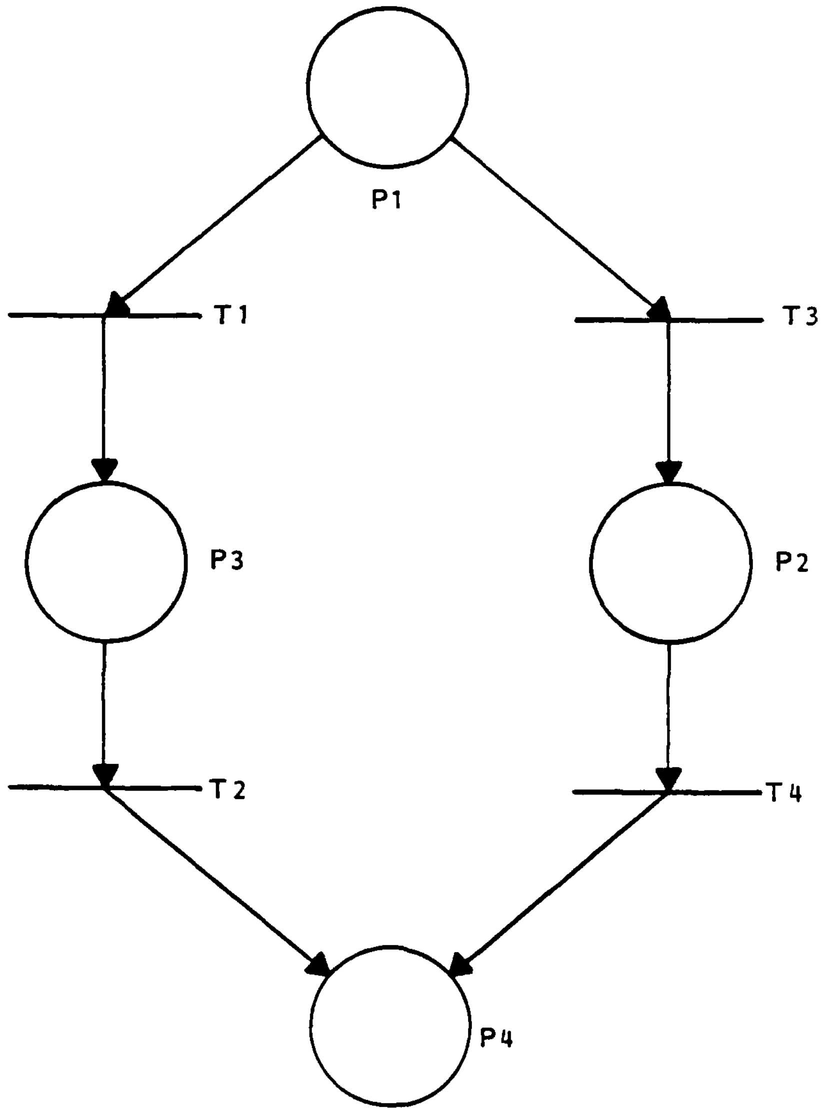


Figure 8.3 Petri Net Representation of an "IF" Statement

8.5 TEMPORAL BEHAVIOR ANALYSIS WITH AP NETWORKS

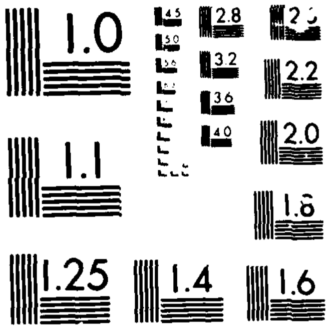
One of the interesting Petri net based modeling techniques is the Abstract Process network (AP-Net) technique. This network technique starts with a subset of Petri nets, and then extends it to represent structured software design better. The resulting network representation has an associated algebra for expressing the execution sequence of various program elements directly. The way the network is built solves many of the problems of modeling timing performance with general Petri nets. Among the improvements is a better mapping between the network and the software structure and a simple, unambiguous method of decomposing network elements. Another improvement is that AP networks are easy to build. The difficulty of building an AP network is on the order of the difficulty of building a structured flowchart.

AP network modeling is specifically aimed at the detailed design and coding stages of program development. This area is where the flow of control has the most influence on timing performance. Other modeling techniques are better suited for other phases of the program development.

AP networks form an extended subset of Petri nets that provides a representation of the flow of control through a program, as described in [28]. The Petri net concept of a transition is extended to include two different types of transition: a state definition transition and a state transformation transition. A state definition transition is limited such that it has either one input place and n output places (state analysis transition), or n input places and one output place (state synthesis transition). State definition transitions are used to show nonsequential relationships between pieces of code, such as iteration, recursion, conditional statements, process forking, etc. Figure 8.4 shows an example of a simple AP network with two state definition transitions used to show iteration. A state transformation transition has a single input place and a single output place. This type of transformation operates on system data. All time consumed in the system is assumed to be consumed in this type of transition. AP network places are limited to one input and one output. They represent the intermediate state of a computation in between computational steps which are represented as transitions.

AD-A169 687 DESIGNING REAL-TIME SYSTEMS IN ADA (TRADEMARK)(U) 2/3
SOFTECH INC WALTHAM MA 1986

UNCLASSIFIED F/G 9/2 NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6

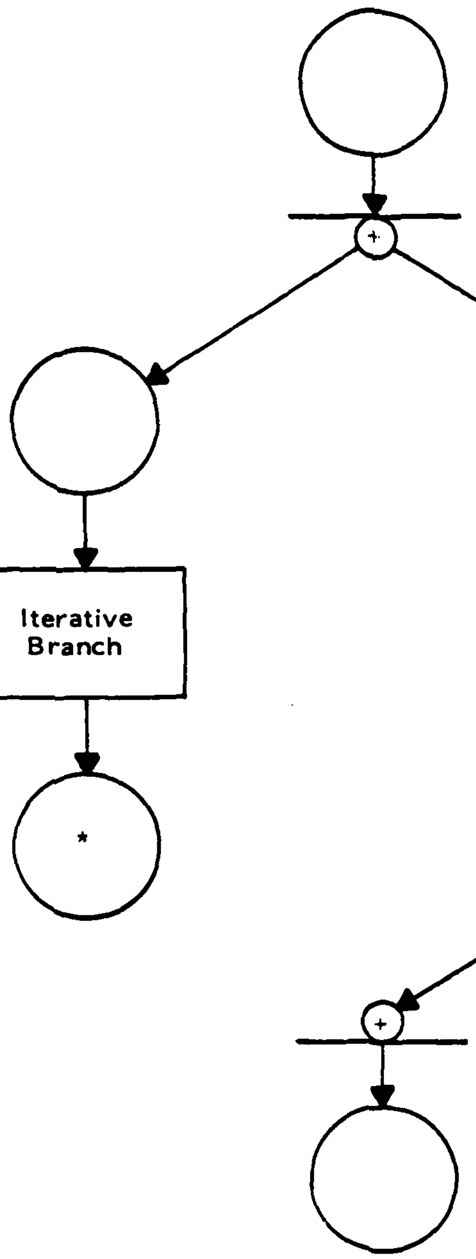


Figure 8.4 AP Network of an Iterative Loop

The form of the AP network is designed to model software. It combines the modeling power of a Petri net subtype with ideas from structured flowcharts. The result is a very powerful and well structured model for software performance. Figure 8.5 shows how several important constructs are represented in an AP network. The major drawback of AP networks as described in [28], is in the area of data communication. The AP network models the flow of control through a system. This is fine as long as data is always passed along with the flow of control (as program parameters or as returned results). There are situations where other methods of communication may be necessary. This is especially true in the realm of interprocess communication. In this area there needs to be some model of common communication techniques such as message passing, queueing, and rendezvous. Mekly [28] suggests that such communication be modeled using standard Petri nets. Unfortunately this makes the process of evaluating the performance of the model much more difficult. A better approach might be to add data communication connections which model data connections as a second type of arc and severely limit the ways in which they can be added into a graph.

8.6 PROCESS LEVEL MODELING TECHNIQUES

Attention has been focused recently on what we will call process level modeling. A model is built consisting of communicating processes. Analysis is performed which depicts system performance as a function of the data flowing through the system. One instance of this type of model is the data flow model. It is important to note the difference between these modeling techniques and the AP network technique. The AP network model concentrates on the flow of control through a program and is weak in terms of representing the effects of data flow. Process level models represent the flow of data but make no attempt to model the flow of control through the program.

Process modeling techniques are suitable for the middle stages of program development (high level design). There is, however, a point beyond which it makes little sense to decompose the process level model further. This point marks the end of the high level design; another modeling technique (such as AP networks) will be necessary for subsequent decompositions of the design.

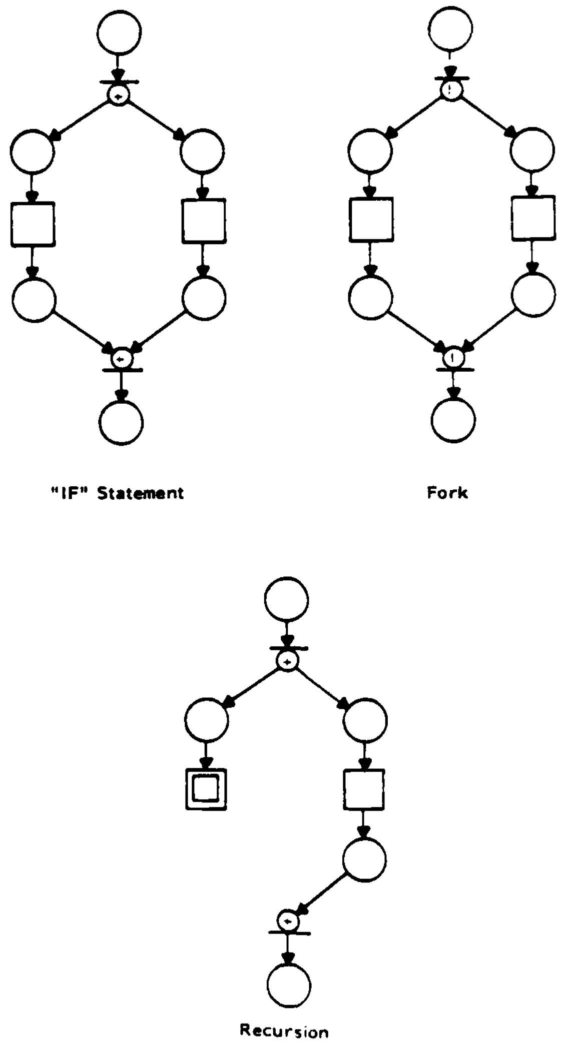


Figure 8.5 Common Constructs

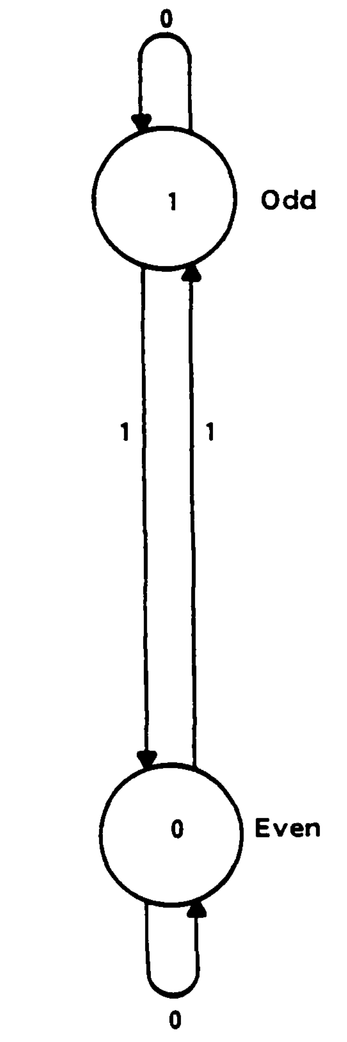
8.7 OTHER MODELING TECHNIQUES

Although Markov chain modeling can be shown to be equivalent to stochastic Petri net (Petri nets with random variables used to represent the statistical nature of time consumption) modeling, there are some properties of this representation that are superior. The major reason for this is that the Markov chain representation can often be simplified, thus making the representation easier to manipulate. This simplification process, however, adds significantly to the difficulty in creating the model.

Finite state automata (often called finite state machines) model software as a set of states and transitions between those states when certain events occur in the system. These models are directly representable by a subset of Petri nets known as finite state machines. This subset of Petri nets limits all transitions to one input place and one output place. These places correspond to states and the transitions to state transitions. Figure 8.6 shows a simple finite state automata and its corresponding Petri net finite state machine. This finite state machine receives a string of bits and determines whether the the number of set bits was odd or even.

Computation Structures [44] use two diagrams to represent the software design: a data flow graph and a precedence graph. The data flow graph shows the relationship between the data maintained in the system and the operations that can be performed on that data. This graph consists of three types of elements: cells, operations, and decisions. Cells provide storage for data, operations transform data, and decisions evaluate a predicate based on some data and send the result to the precedence graph. The precedence graph shows the order in which operations in the data flow graph are to be fired. This graph is composed of a start node, an end node, operation nodes, "and" nodes, "or" nodes, and decision nodes. The start and end nodes are self explanatory. Operation nodes correspond to the operations shown in the data flow graph. "And" nodes allows execution to continue only after inputs have been received from all input lines. "Or" nodes allow execution to continue after an input has been received from any input line.

It can be shown that the modeling power of the computation structure approach is equivalent to that of Petri nets. This type of representation (separating data flow from the flow of control) provides advantages and disadvantages. In some ways the separate graphs make the design more understandable since data and control flow cannot be confused. In some ways, however, this approach is less understandable since it is now necessary to examine and understand two graphs and the interactions between them. It is not clear which approach is the most advantageous.



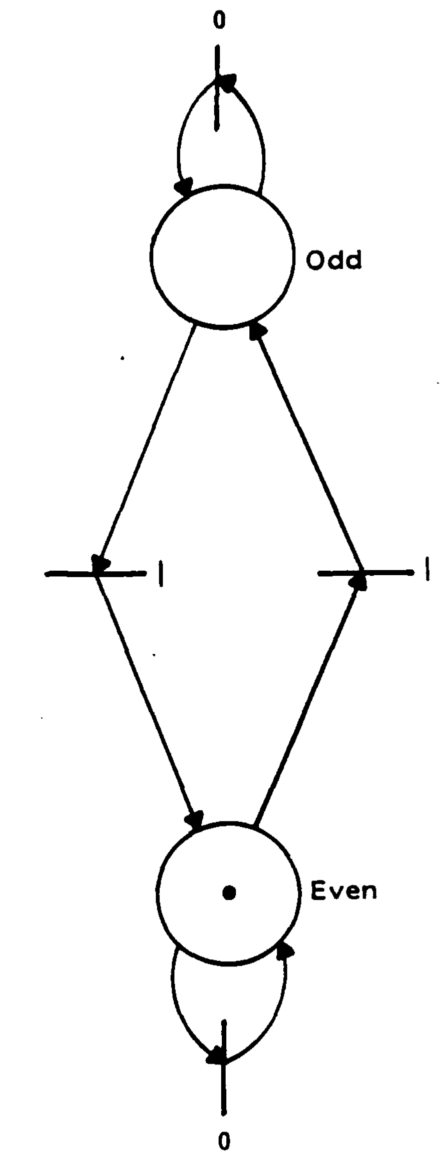


Figure 8.6 Equivalent Finite State Representations

8.8 ANALYZING THE MODEL

Once a system has been modeled it is necessary to evaluate the model in order to gain information about the timing performance of the system. The two major categories of methods for estimating timing performance are simulation and analysis. Simulation determines system performance by simulating the execution of various paths through the software. Analysis uses some methods of generating mathematical statements that describe the behavior of the system. Simulation is relatively easy to do but may not detect some of the important aspects of system timing. Simulation also has the tendency to overwhelm the designer with information, sometimes causing him to miss small but important pieces of data. Analysis can be extremely difficult to perform. In large systems the interactions of various components may be so complex that analysis is not practical.

Evaluation of large complex systems may benefit from some combination of simulation and analysis techniques. Analysis can be used to select a set of simulation paths that will result in a good understanding of timing behavior. Simulation can be used to develop approximate models for parts of the system that are less critical, thus simplifying the model which must be analyzed. By using these techniques in concert it becomes possible to evaluate the performance of many systems which could not be evaluated with either technique individually.

8.9 COMBINING TYPES OF ANALYSIS

Unfortunately, no single method of timing analysis can meet all the needs of a real time designer. The network (such as AP-network) models offer the greatest breadth of modeling since they can be used for both statistical and deterministic modeling. Their major weakness is that they require a more detailed design of the system than is available in the early stages of design. A single timing model also tends to emphasize one aspect of the system at the expense of the others. For example AP-networks emphasize the flow of control, and process level models emphasize data flow. These two factors together point to a modeling system based on an integrated set of different modeling techniques.

Through evaluation of the various techniques discussed in this chapter a set of techniques has been identified that could be combined to form a highly effective modeling system. This set of techniques includes: queueing models, process level models, and AP network models. The queueing models would be introduced during initial system specification. These models would define the expected system resources and specify initial expectations of system performance. Process models would be defined during high level design. These models are analyzed to verify that performance meets the requirements defined in the queueing model as well as any new requirements. The information from the process level model will also be used to refine the workload characterization of the queueing model. During detailed design an AP network model will be generated. This model will be used to check timing performance against expectations, and to refine the numbers in the previous two models. Timing requirements will be introduced during all phases of the development process. These requirements will be introduced into the timing model which is most appropriate for the particular requirement. These requirements are then propagated to the other models. The final modeling step will be the automatic generation of an AP network model from the finished code. This model will be used to verify timing requirements and to further refine the other models.

The different models present after the development of a software system in this manner each provide an independent view of the system timing. The refined queueing model shows the relationship between the software system and critical system resources. The process level model shows the data flow through the system. The AP network shows the flow of control through the processes. Together they form a very detailed view of system performance.

8.10 CONCLUSION

We have examined several methods of analyzing the timing performance of a software system. They vary considerably in terms of how they represent time and how they are used to predict timing performance. By combining a number of these methods into a single integrated method it seems possible to learn a great deal about the timing of a real time system. It is also possible to follow system timing (to the degree that it can be known) from the earliest stages of design right through the coding process. As the design becomes more detailed the timing measurements become more deterministic.

However good a performance analysis tool is, it can never measure the timing of all software on a deterministic basis. This deficiency can be overcome by combining analysis and simulation techniques in order to set limits on the range of possible variation. The designer can then focus on the task of having the range of possible timing paths fall into the system timing constraints.

Figure 8.7 shows a comparison between the requirements for temporal analysis and the performance of a potential integrated tool involving queueing networks, process level models and AP nets. A comparison of each of the techniques discussed to the requirements is shown in appendix A. More work is needed to define an integrated modeling tool along the lines described in the previous section.

1) Temporal analysis must be available at every step in the design process from inception to code.

The integrated modeling techniques provide support over a large part of the design. Queueing models provide support for initial sizing estimates and load distribution. Process level models provide the basis for the decomposition of the design through the high level design stages. AP networks with deterministically or statistically represented timing support the detailed design and coding stages.

2) The analysis technique must be capable of expressing timing in either statisical or deterministic terms, or possibly some mix of the two.

The queuing network supports a highly abstract statistical model of the hardware and software in the system. The AP network and process level models can support either a statistical or deterministic representation of software timing at either a high or low level of detail.

3) The analysis technique must be able to find answers for a variety of questions, including:

- Does the system meet a particular constraint?
- What is the measurement of system performance versus that constraint?
- What parts of the system influence that measurement?

Each of the three models can provide some information in answer to each of these questions with regard to the measurements that make sense in the respective model (see appropriate charts in appendix B).

Figure 8.7. Can an integrated set of Techniques fulfill the requirements for temporal analysis? (Based on a postulated system including queueing networks, process level models, and AP networks.)(1 of 2)

4) The analysis technique must support a variety of timing measurements, including:

- Maximum Response Time
- Average Response Time
- Average Jitter
- Maximum Jitter
- Throughput
- Processor Utilization

Some support is provided for each of these measurements. For details see appendix A.

Figure 8.7. Can an integrated set of Techniques fulfill the requirements for temporal analysis? (Based on a postulated system including queueing networks, process level models, and AP networks.)(2 of 2)

CHAPTER 9

A REAL TIME SYSTEM IN ADA: CASE STUDY 1

The first part of this study has described several methods of approaching the requirements addressed by cyclic executives using Ada. In this chapter we code a sample problem that traditionally would be approached by the use of a cyclic executive. The problem is coded both as a cyclic executive and as a data driven design. The advantages and disadvantages of the resulting code are then analyzed with respect to each of the proposals discussed in Chapter 6.

9.1 REQUIREMENTS DESCRIPTION

The problem selected for this example is a program that extracts test data from a radar system. There are five areas of requirements for this system. They are:

[1] Data Extraction

[2] Data Recording

[3] Operator Input

[4] Status Displays

[5] Interface Processing and Hardware Testing

[1] Data Extraction - The data to be extracted falls in three categories: FIR (Finite Impulse Response) data, Contact data, and Event data. FIR data can be up to 32K words long and consists of intermediate data from the radar's signal processing pipeline. Contact data consists of the ordinary outputs of the radar, showing the position of targets, jammers, etc. and can consist of up to 2K words. Event data are ten words which are counters of important events taking place in the radar returns, such as threshold crossings, etc.

Each category of data is collected for a portion of the surveillance area. This portion is called a gate for that type of data. Each gate is defined by a start range, a stop range, a start bearing, and a stop bearing. Any data (of the type collected by the type of gate defined) that falls into the area defined by the gate parameters will be available for extraction. Each type of gate has a limitation on its size. Event gates can cover any range and up to 359 degrees. Contact gates can cover any range and up to 90 degrees. FIR gates can cover up to 30 miles by 30 degrees.

[2] Data Recording - Data are recorded on two media, disk and tape. Up to eight consecutive scans of FIR data can be recorded on a single disk. One hundred consecutive scans of contact and event data can be stored on a single tape. The disk can record data at the rate of 3200 words per second. The tape can record data at the rate of 667 words per second.

[3] Operator Input - Gates are inserted when commanded by the system operator. Up to one gate of each kind may be activated during each scan. Since the radar in question rotates at 12 rpm, each scan is approximately 5 seconds long. Each gate is controlled through a piece of hardware called a gate controller. A gate's range and type are specified by placing the appropriate values into hardware registers connected to the gate controller to be used. When the radar reaches an azimuth such that the next azimuth to be seen will be within the range of the gate, the gate is activated by setting an active bit in the hardware registers connected to the gate controller. When the stop bearing has been reached, the gate active bit is cleared to the gate controller. Gates inserted by the operator will be repeated every scan until the operator cancels them.

In a similar manner, up to two test targets or clutter targets may be inserted into the radar system per scan. Targets inserted by the operator will be repeated until canceled. Provisions must be made so that scan to scan target motion can be added to the system in the future.

[4] Status Display - Capability must be provided to display information to the operator upon his command. The following displays provide all the optional status information:

- Radar Status Display
- Menu
- FIR Control Word Display
- Contact Data Display (up to 6 contacts and 2 jams)
- Event Count Display

[5] Interface Processing and Hardware Testing - Data between the radar and the program is exchanged across 3 interfaces. Event counts are read from 10 hardware registers that look just like memory to the program. Contacts are sent from the radar through a hardware FIFO. While the contact gate is open the FIFO should be checked at least every 1.5 degrees and any data it contains should be extracted. FIR data is extracted across a special 16 bit I/O channel. The protocol required to operate this channel as well as the procedures used to access the other data are described in Appendix E. In order for the FIR interface to remain operational a test pattern is exchanged with the radar every five seconds. If the test data is not provided to the radar within the time expected, the radar will disable all inputs from this program and will not supply any extraction data until the test pattern exchange is reinitiated.

System data flow is shown in Figure 9.1.

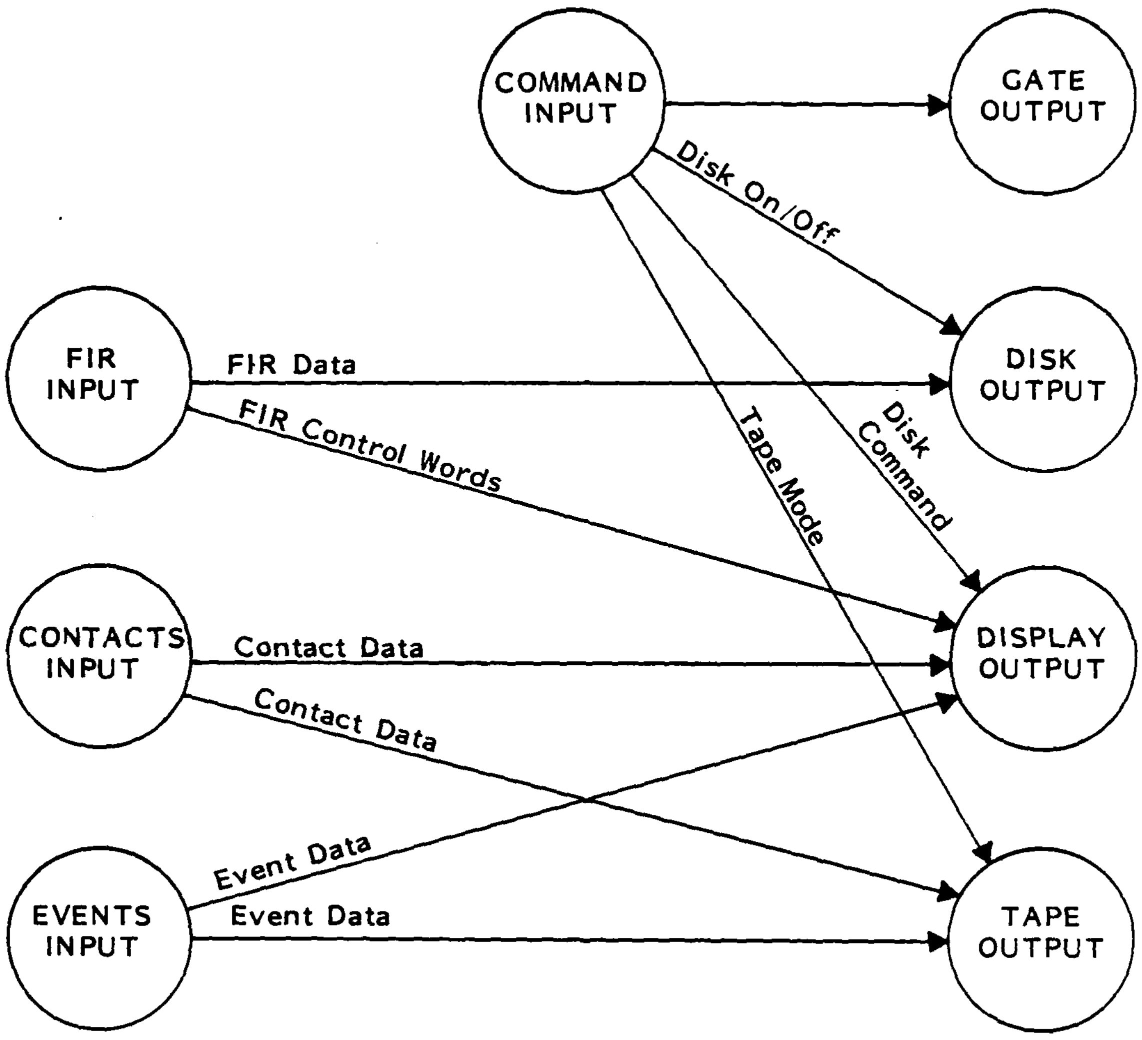


Figure 9.1. System Data Flow.

9.2 PROGRAM DESCRIPTIONS

Shown in Appendix F and G are two solutions to the sample problem. Appendix F shows a solution using a cyclic executive approach to the problem. Appendix G shows a solution employing a data driven design approach. The cyclic executive approach is representative of cyclic executives driving Ada code (proposal class 1 from chapter 6), while the data driven design is representative of the output of the Data Driven Design Methodology (proposal class 2), and the input to the Transformational Tool (proposal class 3).

The cyclic executive solution is a degenerate case where only one minor cycle is needed per major cycle. This cyclic executive has been implemented as the simplest possible type with no capability even to detect overruns, let alone make compromises if they occur. In order to use this type of cyclic executive, the system designer must verify that there will be no overruns before the software is run.

The cyclic executive solution is designed to resemble the code that was originally written to solve the problem that this exercise is based on.

The cyclic executive approach uses Ada's multi-tasking facilities to support background, cyclic, and interrupt driven tasks (see Figure 9.2). A task is used to represent each of the minimal concurrent processing elements. The use of tasking features greatly simplified the coding of the cyclic executive system. These tasks communicate through the use of shared data. Priority is used to specify which task is allowed to preempt other tasks (i.e., the background tasks are given lowest priority, the interrupt tasks the highest priority (except for keyboard handler), and the cyclic task an intermediate priority. Thus the background runs only when no cyclic or interrupt processing is needed). Processing is organized in packages (i.e., all cyclic tasks are in one package, etc.). Because entries are executed at the priority of the caller, interrupt entries are executed very quickly. In some cases this will cause interrupt driven tasks (such as keyboard handler) to become ready to run before they can perform any processing due to their lower priority.

The cyclic executive itself is coded with only an eleven line task body as shown in Figure 9.3. It simply waits for an interrupt and calls the routines in the frame. In addition to being easier to code, the Ada implementation of the cyclic executive should be far easier for a maintenance programmer to understand than the typical cyclic system. The typical cyclic executive requires a maintenance programmer to either wade through several hundred lines of assembly code (often of notoriously bad structure), or several system manuals to understand the program/operating system interaction inherent in the design. In this example Ada cyclic executive, it only takes knowledge of the Ada language.

```
package Event_Driven_Package is           -- Package Containing High Priority
                                          -- Event Driven Processing

  task Batch_Handler is                   -- a batch occurs approximately every
    entry Batch_Interrupt;                -- 20.8 msec (or every 1.5 degrees)
    for Batch_Interrupt use at 8#100301#;
    pragma priority (10);
  end Batch_Handler;

  task FIR_Handler is                     -- FIR interrupts occur when FIR data
    entry FIR_Interrupt;                  -- is ready to be extracted.
    for FIR_Interrupt use at 8#100302#;
    pragma priority (9);
  end FIR_Handler;

  task Disk_End is                        -- This interrupt occurs when a DMA
    entry Disk_Interrupt;                 -- transfer completes.
    for Disk_Interrupt use at 8#100303#;
    pragma priority (8);
  end Disk_End;

  task Keyboard_Handler is                -- This interrupt occurs whenever a
    entry New_Character;                  -- key is pressed on the keyboard.
    for New_Character use at 8#100304#;
    pragma priority (4);
  end Keyboard_Handler;

end Event_Driven_Package;


package Cyclic_Routines is                -- Package of all processing that
                                          -- runs at the standard cyclic rate
  task Cyclic_Executive is
    entry Minor_Cycle_Start;
    for Minor_Cycle_Start use at 8#100300#;
    pragma priority (5);
  end Cyclic_Executive;

end Cyclic_Routines;


package Background_Routines is            -- Package of all processing that
                                          -- runs in background.
  task Background_Task is
   pragma priority (2);
  end Background_Task;

end Background_Routines;
```

Figure 9.2. Package specifications for Cyclic Executive system

```
task body Cyclic_Executive is               -- This Executive calls the
begin                                       -- routines in a minor frame for
  loop  -- forever                          -- each five second interrupt.
    accept Minor_Cycle_Start;
    Interface_Test;
    FIR_Extraction;
    Disk_Storage_Initiation;
    Target_Output;
    Tape_Output;
  end loop;
end Cyclic_Executive;
```

Figure 9.3. Cyclic Executive Task Body

It is useful to note that within the cyclic executive packages most inter-routine communication is carried out through the use of shared data reflecting the original design of the system on which this example is based. (Much of this shared data could be eliminated through the use of parameter passing within the cyclic package.) The data passing mechanism utilizes a new-data flag for each set of data passed between tasks. The producing task, when it wishes to output data, first checks to see that the new-data flag is clear. If the flag is clear, the data can be written into the buffer and the flag is set. If the flag is set to begin with, the data cannot be written into the buffer. The consumer, any time it sees the new data flag set, will read the data in the buffer and clear the new data flag. For example, when gates are sent from command processing to the routine that opens and closes gates the flag New_Background_Data is used to coordinate the transfer.

Data flow in the cyclic executive system is shown in Figure 9.4.

The Data Driven design approach is an example of one possible approach that fits the data driven design concept. All real time constraints are placed on the system inputs and outputs with the assumption that if the inputs and outputs are meeting the system timing requirements then the software must be meeting them too. In this design each data buffer is encapsulated in a task, which also contains the description of what is to be done with the data buffer. Tasking in this example is used to express functionally independent software responsibilities, not the time line to be followed during execution. Processing is organized in packages of procedures and tasks by function, not by timing criteria (e.g., the software needed to extract FIR data is organized into its own package, not combined with other software to form a cyclic executive package). This promotes greater functional modularity.

The data flow for the data driven design is shown in Figure 9.5.

In both designs there are many things that must be done before the design and analysis of the system is complete. First, system timing requirements must be specified in a consistent and readable fashion. Second, the timing performance of each solution must be analyzed in detail and compared with the requirements. Third, potential timing trouble spots must be identified in the design. Finally, work-arounds for each trouble spot must be developed so system timing requirements are met. It is in these areas that methods and tools must be used to supplement the Ada code. By comparing the approaches available to support each proposed methodology to address these items, we hope to gain insight into how each methodology can be used. Additionally, we might postulate on possible improvements that might be made by the use of these methods.

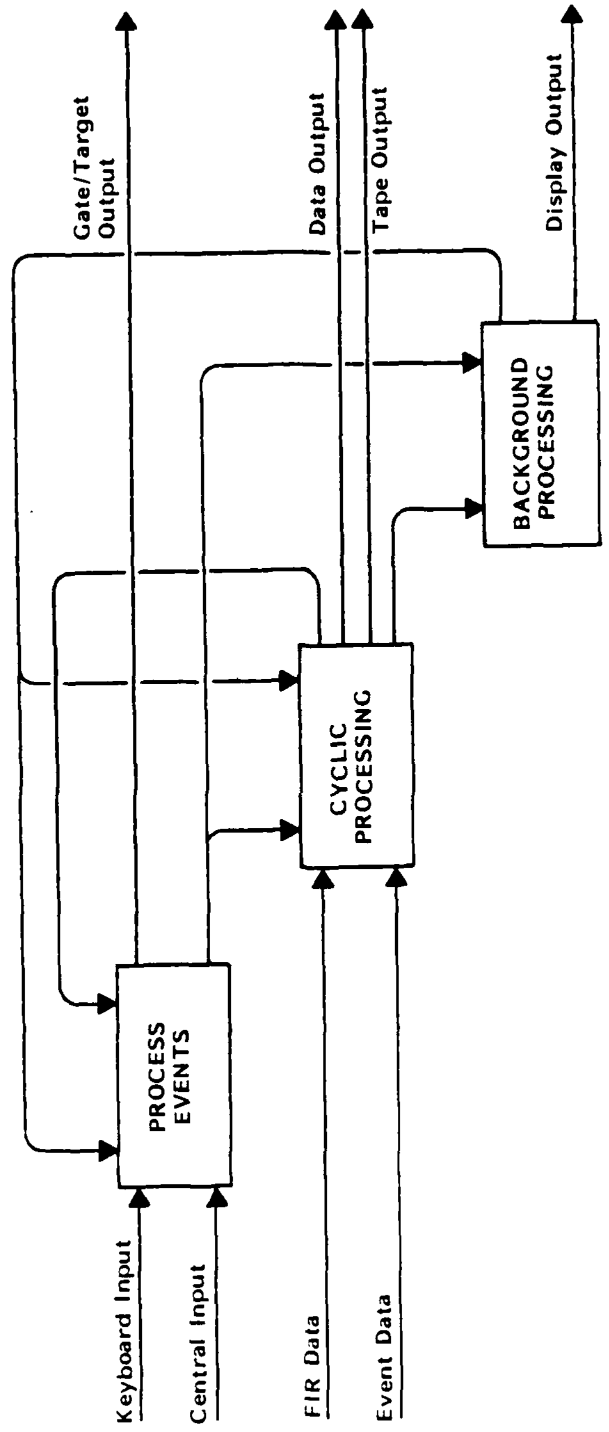


Figure 9.4. Cyclic Executive Data Flow

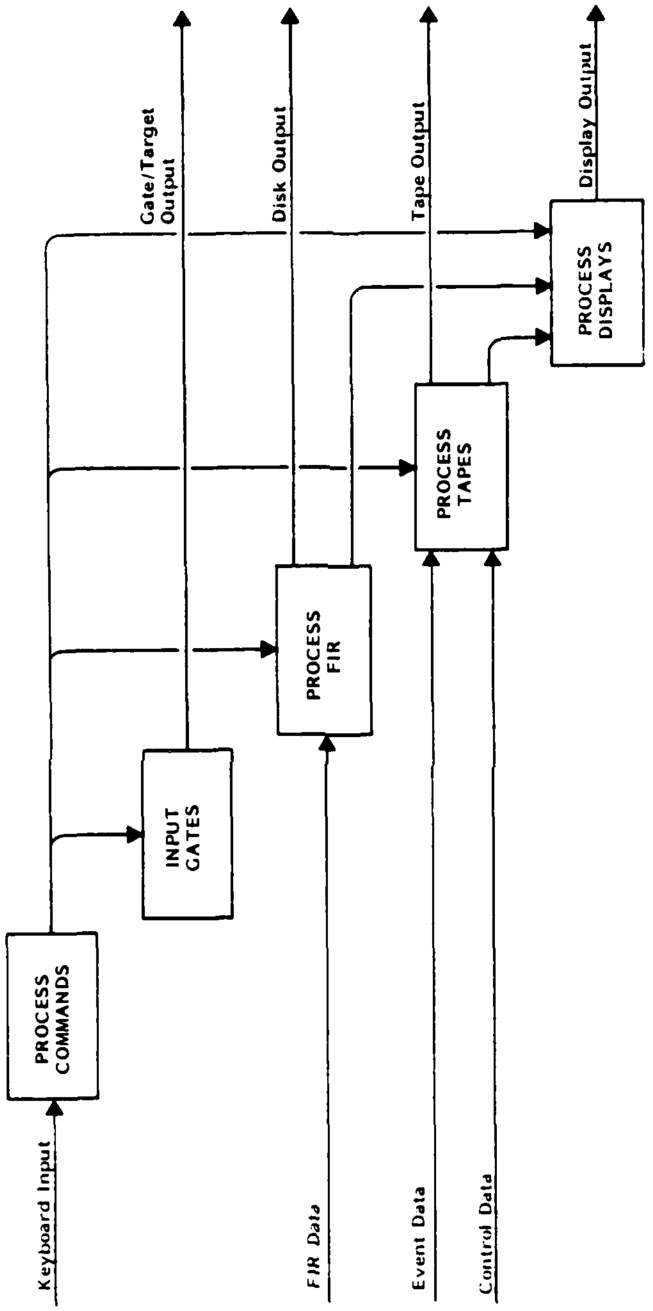


Figure 9.5. Data Flow for Data Driven Design

9.3 DETAILED PROGRAM DESCRIPTION

In order to provide a better understanding of the workings of each of the implementations we will describe one function in detail for each implementation. The function to be described will be the FIR extraction and recording function. This function extracts 32768 word buffers of data from the hardware and records them on a disk. Five words of each buffer are also saved for a potential display. Operator commands can turn the disk on or off. If the disk is off, the data is extracted only to be sent to the display.

In the cyclic executive design the following modules are involved in the FIR extraction and recording process:

```
Cyclic_Package
  Cyclic_Executive
  FIR_Extraction
  Disk_Storage_Initiation

Background_Routines
  Command_Processing
  Display_Processing

Event_Driven_Package
  Disk_End
  FIR_Handler
```

When FIR data is ready for extraction, an interrupt is received by the FIR_Handler task in the Event_Driven_Package. FIR_Handler sets a flag (Data_to_Cyclic.FIR_Ready) to the FIR_Extraction procedure in the Cyclic_Package to inform it that there is data to extract. The FIR_Extraction procedure is called every cycle by the Cyclic_Executive task. If there is no data to extract (FIR_Ready = False) the routine returns without doing anything. If there is data to extract FIR_Extraction must first verify that there is a buffer to put the data into. This is done by making sure the Number_of_Buffers_Active counter is less that four (there are 4 buffers). If there is room, the data is read into the next buffer (pointed to by Input_Buffer_Number) by the Low_Level.FIR_In procedure. The FIR control words (16380..16384) are then copied into the display buffer (Data_to_Background.Control_Words). Finally, Input_Buffer_Number and Number_of_Buffers_Active are both incremented to indicate that another buffer has been filled.

Data is written to the disk by the Disk_Storage_Initiation procedure. This procedure is called by the Cyclic_Executive task each cycle. The first thing that this procedure does is to check whether a disk I/O operation has completed (indicated by the Disk_is_Done flag). If a disk operation has completed Number_of_Buffers_Active will be decremented to indicate the new free buffer. If there is data to be written and the disk is not already active, Disk_Storage_Initiation will try to output the next output buffer. If the disk is on, a DMA transfer will be started to move the next output buffer (pointed to by Output_Buffer_Number) to disk. If the disk is off, output will not be started, but in both cases Output_Buffer_Number will be incremented to point to the next buffer. If the disk is off, Number_of_Buffers_Active will be decremented to indicate that the buffer can be reused. When a disk operation completes Event_Driven_Package.Disk_End receives an interrupt. It then informs Low_Level that the disk has finished and sets the Disk_is_Done flag to the Disk_Storage_Initiation procedure.

When Display_Processing needs FIR data it can get what it needs from the Control_Words buffer in Data_to_Background. This buffer always contains the latest control words received.

Command processing controls whether the disk is on or off by using the Disk_On flag in Data_to_Cyclic. If this flag is true, data will be recorded on the disk, otherwise it will not be recorded on disk.

In the Data Driven design the following modules make-up FIR extraction and recording:

FIR_Recording

Display_Processing

Command_Processing

Display_Processing retrieves the latest FIR Control data by calling FIR_Recording.Get_Control_Words.

Command_Processing can turn the disk on or off by calling either FIR_Recording.Disk_On or FIR_Recording.Disk_Off.

The rest of the work is concentrated in the package FIR_Recording. This package is composed of eight tasks which together perform the FIR_Recording duties. FIR_Data_Ready_Handler keeps track of the state of the FIR interface and whether or not there is data to extract. Its mission is to coordinate entry into the FIR interface between the Extraction_Tasks and possible interface tests. Only one task is allowed into the interface at a time, and Extraction_Tasks are only allowed in when there is data ready to be extracted. All tasks check out with the FIR_Data_Ready_Handler when they leave the FIR interface.

The Disk_Access_Controller keeps track of the state of the disk. Only one task is allowed to be using the disk interface at a time. When a task finishes using the disk it checks out with the Disk_Access_Controller.

Display_Buffer maintains a copy of the latest FIR control words for Display_Processing. This data is made available to Display_Processing through a procedure that is visible to the world.

Transfer_to_Disk performs the actual disk output. The task that calls Transfer_to_Disk to store data will remain in the entry call until the transfer is complete. When the disk is not on, Transfer_to_Disk accepts entries but does not start disk I/O. Instead it lets the entry finish as if the disk I/O were complete.

The four Extraction_Tasks are agent tasks that perform the extraction and recording operation. An Extraction_Task first waits until FIR_Data_Ready_Handler tells it that it can extract data from the FIR interface. Data is extracted through a Low_Level I/O call. The control words are copied over tc Display_Buffer. The Extraction_Task then waits until Disk_Access_Controller tells it that it can record its data. When it is told that it can record its data it calls Transfer_to_Disk to perform the actual transfer. When the transfer is complete the Extraction_Task goes back to wait for FIR_Data_Ready_Handler to tell it that it can extract more data.

This design is necessarily more complex than an ordinary buffering scheme. This is the result of two requirements. First, the buffering operation cannot involve copying data due to the large size of the data buffers. Second, the design must be able to input to one buffer at the same time that another buffer is being emptied.

9.4 PROGRAM COMPARISON

We will now compare the two basic approaches (cyclic executive and data driven design) against the requirements for an Ada solution developed in chapter 6. After we have examined these approaches we will discuss how the transformational tool might apply to this problem.

The comparison of the two approaches will be made by first evaluating them against the requirements traditionally addressed by cyclic executives and then against the problems caused by the traditional cyclic executive approach. The requirements traditionally addressed by the use of cyclic executives include:

- A method of fulfilling timing requirements including periodic, jitter, maximum execution time, and maximum time between executions.

- A method to schedule processing without incurring high overhead costs.

- A method of rapid overload detection.

The ability to fulfill all the timing requirements mentioned is not an obvious property of either solution. In each case the ability of the solution to fulfill a particular timing requirement is a combination of the capability for that timing requirement to be met by that type of system, and the capability of the associated methodology to satisfy the requirement with a coding strategy and verify that it is being satisfied in the resulting solution. For periodic requirements it is easy to see that in the cyclic design, as long as all cyclic processing fits in its frame, all periodic software will run at the prescribed rate. The same statement can be made about the data driven design if only one input and output are considered; if all the necessary processing fits into the interval between input and output, all the periodic processing that works on that data will be performed at the proper rate. The only differences are that in the cyclic case the order of that processing is clearly deterministic, and the amount of time spent scheduling tasks is lower. The data driven design may follow several different execution orders depending on the scheduler, and more time will be consumed scheduling tasks. When several input and several outputs are introduced the problem becomes more complex for the data driven design approach. The timing of when data enters and leaves the system is variable. Each pattern of input and output timing will yield a different set of possible scheduling paths. As one timing path follows another an input or output timing constraint can easily be violated. The cyclic executive, on the other hand, maintains strict timing relationships between all inputs and outputs so their relative timing cannot cause interference.

In the designs shown in this chapter the potential of inter-routine interference is most obvious between the software that records FIR data and the software that records data to tape. In the cyclic executive case each of the routines in these functional areas has been assigned a time slot in which to run. As long as none of these routines overruns its assigned time slot there can be no difficulty. In the data driven design case the situation is different. There is little problem for the data driven design to find a good time line for either one of these extraction and recording missions, but for both of them is more difficult. It is not too difficult to imagine a situation where tasks from both sets are competing for the processor due to data for both extractions arriving at the same time. How scheduling is accomplished becomes critical to the success of the mission. The scheduler would have to be able to pick tasks that are needed to meet the next deadline. In many cases the tasks involved are only indirectly related to the task upon which the deadline has been placed. The resulting schedule will most likely sometimes miss deadlines.

A timing requirement that might be reasonable to impose is a minimum time between executions for command processing. Obviously the operator should not have to wait forever before his commands reach the system, so this is a fairly reasonable requirement to impose. In the cyclic system this can be approached by computing the percentage of the processing resource that is consumed by the cyclic and event driven software during various operations. The time between executions of the command processing routine would then be the length of time it would take to execute the command processing routine and the display processing routine divided by the fraction of the processor available to run them. The time line of the cyclic executive is shown at the beginning of

Appendix F. If the execution times of the blocks shown are known, we can compute the percentage of the cycle used. If we then subtract out the maximum allowable incursions from the event driven tasks, we obtain a measure of the percentage of the cycle available to the background tasks. The maximum time between executions is now fairly easy to compute.

In the Data Driven Design the maximum time between executions constraint is far more difficult to verify. If we assume a preemptive scheduler which picks highest priority first and longest-ready-to-run if everything is at the same priority, then we can find the maximum time between executions by adding the execution time of all the routines with higher priority or the same priority as the command processing routine together with the highest possible overhead for scheduling those routines. This sum would tend to yield a longer time than the cyclic executive maximum. The data driven example lacks the simple structure of the cyclic executive as there is no cycle that can be used to compute a percentage of processor utilization. In this particular problem it is not difficult to use knowledge of the application to do a similar analysis as that done on a cyclic executive: it is the result of all the software being excited at the same periodic rate. In a system where the excitation of the software in the system does not follow such a simple pattern, the computation of a percentage of processor utilization can be very difficult.

The minimum time between executions in the data driven design case is just the run time of command processing. It seems unlikely that the cyclic executive case can match that.

Verifying jitter requirements in the cyclic executive is simply verifying that no overrun will occur and computing the maximum impact of event driven processing. Jitter in the data driven example involves computing the execution time of all routines of higher or equal priority than the subject routine that can possibly execute before the subject routine is triggered, plus the overhead involved in the scheduling decisions involved in running them. If in the example a jitter requirement had been laid upon the periodic execution of the interface test routine, it would not be difficult to verify it in the cyclic executive. The only possible causes of jitter would be a frame overrun or the encroachment of event driven processing. The maximum delay due to these possible interferences is not difficult to compute. Once that is computed it can be compared to the jitter requirement to determine whether or not it will be met. If it is not being met either potential overruns will have to be limited, or the incursion of event driven software will have to be limited. The Data Driven design presents a more difficult problem. The time that data arrives for each of the periodic functions within the period is not known. There is nothing to prevent data from arriving just before the interface test is due. In this case the only way to ensure the jitter requirement is to raise the priority of the interface test above that of the rest of the software. If this change is done, there might be an impact on the requirements of the rest of the software.

Maximum execution time for a module in the cyclic executive system is the execution time of the module plus the maximum legal incursion of event driven processing. Maximum execution time in the Data Driven design is the execution time of the module plus the execution times of all processing of higher priority that could possibly become ready to run during the subject routines' execution. In this coding example, the actual execution times could involve essentially the same routines, depending on the scheduler used for the Data Driven system. The difference is that the data driven scheduler is not constrained to return to an interrupted task immediately after handling an interrupt. It can decide to execute another task of equal priority. If the scheduler were constrained to return to the interrupted task, the execution time would be the same in this case except for scheduler overhead. The important difference is the method of determining the execution time. The cyclic executive case maintains a set of tasks that can interrupt any cyclic routine so the amount of expected interruption remains a constant for all cyclic routines. In the data driven case each task has to be evaluated independently to determine what level of interference is possible.

The overhead of the cyclic executive system is inherently lower than that of the data driven design because scheduling decisions are made at design time while data driven design scheduling is computed at run time.

The cyclic executive shown in Appendix F has little capability to detect overloads. They would not be detected until data structures start overflowing. The data driven design system detects overloads in the same way. The cyclic executive could be modified to detect overruns of the minor cycle, giving a faster indication of overload but no indication of which routine is overloaded. The data driven design would have to put timers on individual routines to give overload indications this quickly.

Solving the problems normally caused by cyclic executives means providing:

- Functional code developed independently from timing requirements.
- Specification of the timing requirements to be satisfied.
- Documentation of the actions taken to satisfy the timing requirements.
- A system that is not fragile when code is changed.
- Greater flexibility than the cyclic executive frozen time line.

The data driven design approach clearly allows the functional code to be far more independent of timing requirements than the cyclic executive. The data driven software is organized by function. For greater readability, each module contains one function. The cyclic executive system is organized by what time the software is supposed to run, with each module being a scheduling system. For example, the cyclic package contains software to extract from three different devices and record the data on two different media, but the software to accomplish this resides in five undifferentiated routines. In the data driven design the processing to accomplish the same functions resides in several well defined packages.

Neither design includes precise documentation of the timing requirements that the system is supposed to meet. The data driven system, however, is organized so that timing requirements are clearer since timing constraints propagate along data paths. This makes the effect of one requirement on the system far more evident than in the cyclic executive case.

Neither system includes any documentation of what actions were taken to meet timing requirements but the data driven design approach has no hidden actions. In a cyclic executive each frame or level assignment could be important to system timing, or it could totally insignificant. It is difficult for someone who did not participate in the executive tuning process to differentiate between important assignments and unimportant ones. If a maintenance programmer wanted to switch the assignments of two routines he would have no idea if the original assignments were necessitated by timing requirements or just happened at random. The data driven design method does not include this type of tuning, so it does not have the same problem.

The cyclic executive is more fragile during code changes than the data driven design. This cyclic executive is perhaps less fragile than many others because its timing interactions are fairly simple (only one frame). Any code change, however, still risks changing the timing interactions of the system. These changes could be as serious as a potential frame overrun or could just be less time for the cyclic routines due to larger event driven routines. Whatever the change in timing there is the potential that it could result in the violation of a timing requirement. This situation is clearer when one considers the way cyclic executives work. A cyclic executive schedules processing to follow a single path of execution. The execution path is optimized to provide the best possible performance. When the timing characteristics of the software change and the timing path does not, the timing path can become non-optimal. The effect is the inefficient use of the processor and the violation of timing requirements. The data driven design constantly adjusts its scheduling path to make efficient use of the processor.

In the area of flexibility the data driven design approach has a clear advantage over the cyclic executive approach due to its ability to adjust the scheduling of the processing of the system to meet the needs of the data flow at any particular moment. The cyclic executive approach is stuck with its single time line unless mode changes are introduced, further complicating its design.

When we compare the operation of the two approaches it becomes clear that they both do the same job in very similar manners. Both systems run at the five second cycle time of the host radar. The difference is that the data driven design approach schedules the software each cycle according to when the data is ready, while the cyclic system runs each piece of software at precisely the same time each cycle. The cyclic executive gains the advantage of eliminating task switching and rendezvous while doing high priority processing i.e., there are no task switches during a frame, only at the beginning and end of the frame to handle background tasks. The data driven design approach on the other hand derives advantages from its greater flexibility and from the fact that its software does not have to be organized according to timing characteristics.

A tool for transforming the data driven design into a cyclic executive combines the advantages of the cyclic and data driven approaches. The transformed code performs like the cyclic executive in fulfilling timing requirements, low overhead, and overload detection. Because the process that generates the cyclic schedule is computer assisted, the work of forming the proper time line is lessened, the requirements that drive the scheduling process are automatically documented, and the actions taken to meet the timing requirements are automatically documented. The source code is developed identically to the data driven design approach, so timing requirements do not impact the source code, only the object code. Fragility is lessened since coding changes can be run through the scheduling tool to check if any timing problems will be caused. The only requirement not improved by transforming the code to a cyclic executive is that of run time flexibility.

9.5 SUMMARY

We have shown two contrasting ways of approaching the problem of real time design. The first, cyclic executives, considers the problem of how to schedule system processing to meet system timing requirements during initial code design. The second, data driven design, leaves scheduling decisions until after the system has been designed and coded. A third method has been introduced which combines the initial data driven design methodology with the cyclic executive run time characteristics by using a transformational tool.

The cyclic executive approach clearly has many weaknesses in terms of modularity, readability, and maintainability. If either of the other two methods offers comparable run time characteristics there seems to be no reason to code a cyclic executive in preference to the comparable approach.

Both of the other two methods have one significant run time problem. Data driven design lacks deterministic timing, and the transformation into a cyclic executive rules out run time flexibility in terms of scheduling to meet the dynamic processing needs of the system. Depending on the needs of the specific application, one or the other of these considerations is likely to be the more important, dictating the choice of scheduling method.

Because the initial software development for each of these proposals is the same, there is already a great advantage to using these two methods. The software can be written before any scheduling decisions have to be made. This advantage can be increased by extending the transformational tool to include timing analysis of normal data driven systems. A set of optimizations could also be developed, such that when applied in a directed manner, data driven designs might achieve enough determinism to meet their timing requirements. In this way systems that do not demonstrate adequate determinism can be modified through optimization and reduction of scheduling freedom in a gradual manner until the correct level of determinism is reached. The ultimate optimization/reduction of scheduling freedom (transformation into a cyclic executive system) is maintained as an option for those cases that require it.

A combined tool of the type proposed would provide a consistent method of real time constraint definition, analysis, and scheduling for any real time system written in Ada. Additionally, the tool would provide automatic documentation of timing requirements and actions, and automatic verification of the suitability of coding changes in terms of timing behavior.

CHAPTER 10

A REAL TIME SYSTEM IN ADA: CASE STUDY 2

Design methods used for real time systems are primarily aimed at meeting the timing requirements of the system. This emphasis leads to a totally different style of programming: organizing pieces of software on a strict timing schedule. This practice, though defensible, leads to highly brittle software designs, high costs, and low reliability. At the other extreme, traditional software engineering methods emphasize the importance of getting the software functionally correct, reliable, simple, and cheap.

With the anticipated increase of complexities in military systems requirements, neither approach is likely to be successful. Thus a necessary first step towards meeting the overall goals would be a more balanced approach to the design of real-time systems. That is, conscious design decisions must be made at each step to ensure that the timing, reliability and reusability requirements will be met. Reusability, we believe, will play a key role in reducing the costs of large and complex systems. Thus far it is largely a novel area and is the subject of some very active research. In this report we take a simple problem and outline a solution that takes into account efficiency and reusability requirements at early stages of the development.

This chapter is structured as follows. In the next section we describe the nature of the problem and give an outline of an efficient solution. Next we describe the notation we use for expressing the design. This notation is an attempt to use Ada as a module interconnection language. In the next few sections we refine the top level design to a detailed level and at the same time describe design decisions that will improve the reusability of each module developed. In the conclusion we describe some of the advantages of the methodology and some areas for further improvements. We also sketch ways in which parts of this design could be reused in other similar applications.

10.1 PROBLEM DESCRIPTION

The problem we consider here is typically found in real-time data acquisition systems, where large amounts of data are being accumulated from a variety of sensors and need to be transferred to a mass storage medium, such as disk or tape. We model this situation as consisting of two processes: a producer and a consumer. The producer corresponds to the sensor process and the consumer corresponds to the disk handler process.

A simple shared data structure, such as a bounded buffer, is inadequate for this purpose for two reasons:

[1] The copying needed to transfer data from the producer to the buffer and from the buffer to the consumer is too inefficient for large chunks of data.

[2] The waits for access to the buffer are affected by the duration of the copying operation. When the copying operation is lengthy, the producer may drop data.

10.2 OUTLINE OF THE SOLUTION

A simple scheme to overcome these difficulties is outlined below. The producer stores data in areas designated by pointers. The pointer, together with its designated storage area, is called a bucket. Each bucket is maintained in a special pool. When data arrives, an empty bucket is retrieved from the pool and filled with data. Then the bucket is put into a queue. On the other side the consumer retrieves a bucket from the queue, drains its contents onto the disk, and finally returns the empty bucket to the pool of buckets. In this scheme, both of the above-mentioned difficulties are overcome: the access times to the pool and queue are limited to the time for copying pointers to and from the queue, and all data copying operations are performed outside the queue when buckets are filled and emptied. In the rest of this report we discuss a detailed solution based on this idea.

With this scheme in mind, we get the following abstract structure of the producer and the consumer processes.

```
Producer:
loop -- forever
-- 1. get an empty bucket from the bucket pool
-- 2. wait for data arrival
-- 3. fill bucket with the data
-- 4. put the bucket into the queue
end loop;

Consumer:
loop -- forever
-- 1. get a bucket from the queue
-- 2. empty the bucket onto the disk
-- 3. put the empty bucket back into the pool
end loop;
```

Figure 10.1 -- Top Level Design

In this design, step 1, performed by Producer, and step 3, performed by Consumer, need mutual exclusion because they both access the common bucket pool. Similarly step 4 of Producer and step 1 of Consumer need mutual exclusion because they both manipulate the common queue. However, the steps of filling and emptying the buckets can be performed independently, thus decoupling the two processes significantly, as the fill and empty operations are time consuming when large amounts of data are involved. Thus the key functional requirements for the Pool and Queue are:

Pool:

[1] Must provide exclusive access to a fixed size Pool.

[2] Must provide capabilities for getting buckets and putting them back.

[3] If all buckets are in use, then the operation for getting a bucket should be delayed until a bucket is available.

Queue:

[1] Must provide exclusive access to a fixed size Queue.

[2] Must provide capabilities for inserting buckets and removing them.

[3] If there are no buckets in the queue, then the operation of retrieving a bucket should be delayed until one is available.

The overall architecture of the system looks like this:

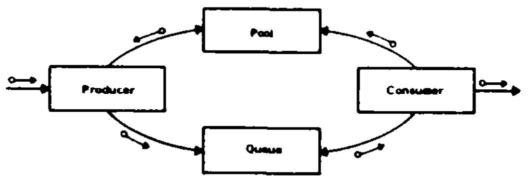


Figure 10.2 -- System Architecture

10.3 MODULE INTERCONNECTION TECHNIQUE

Before discussing the detailed design of the solution we describe the notation used for expressing modular designs. This notation is essentially a module interconnection language. There are two advantages to this notation. First it is based on Ada and hence can be translated to Ada code quite easily. Second, it allows one to observe and understand the detailed connections among various modules that are necessary for achieving any form of reusability. The main ideas are described in [18]. Here we give an overview of the notation with an example.

A module is viewed as a generic package written using as few with clauses as possible. All identifiers to be imported are generic formal parameters. All exported identifiers of this module are described by declarations in the visible part of the package.

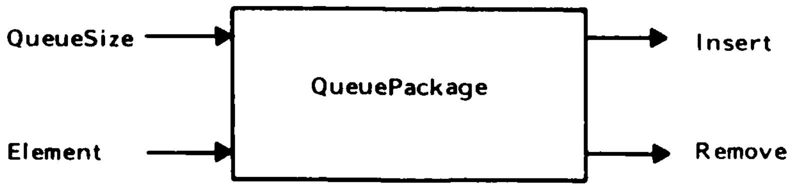
Interconnections among a set of modules are expressed by supplying identifiers exported from one or more modules as actual parameters of some generic package representing a module, in short by generic instantiations.

Pictorially, we can imagine generic packages as boxes with imported identifiers (formal parameters) as arrows directed inwards and exported identifiers (visible parts of the package specifications) as arrows directed outwards.

As an example, the following generic package

```
generic
    Queue_Size : Positive;
    type Element is private;
package Queue_Package is
    procedure Insert(I : in  Element);
    procedure Remove(I : out Element);
end Queue_Package;
```

is represented as:

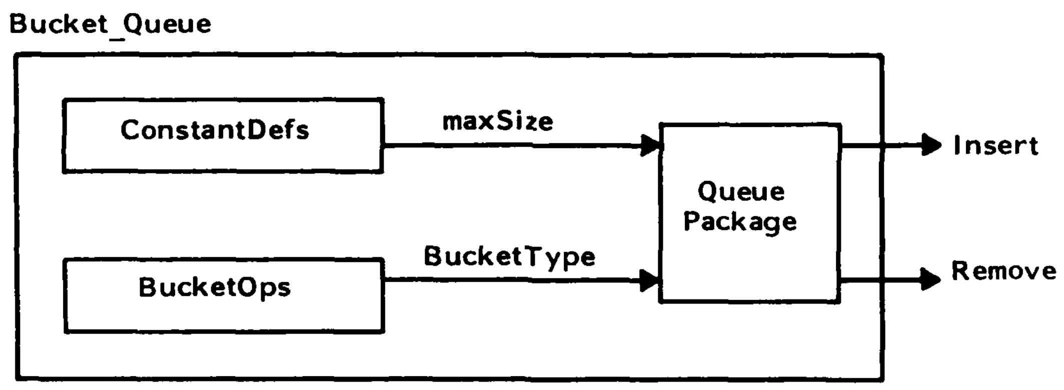


Thus a generic package as written in the above manner specifies all the entities that will be imported from the outside, and the visible part specifies all the entities that will be exported to the outside.

A module interconnection is described by supplying actual generic parameters selected from the export lists of other packages. As an example, consider an instantiation of Queue_Package that gets its definition of Queue_Size from the Constant_Defs package and its definition of the Element type from the Bucket_Ops package.

```
with Constant_Defs, Bucket_Ops, Queue_Package;
pragma Elaborate (QueuePackage); -- * see footnote.
package Bucket_Queue is new
    Queue_Package (Constant_Defs.Max_Size,
                    Bucket_Ops.Bucket_Type);
```

Pictorially this looks like this:



* Pragma Elaborate is needed to avoid getting PROGRAM_ERROR if the body of Queue_Package has not been elaborated. The pragma is not needed for Bucket_Ops since no operations are imported, and so none can be called when the instantiation is elaborated. Constant_Defs presumably does not have a body that needs to be elaborated. On the other hand it would not make the program illegal, and would be fail-safe, to mention all of the packages in the pragma.

10.4 DETAILED DESIGN

Starting from the architectural description of the system as shown in the previous section we present a detailed design. We make conscious decisions to make the design more general and reusable, following some of the guidelines suggested in [5]. Through successive refinements we depict our intermediate design in a diagramatic form that illustrates the module interconnections among various modules. The detailed code is shown in appendix I.

The first step we take is to transform the architectural diagram into a preliminary module connection diagram. This is accomplished by reversing the arrows of calls emanating from various modules. Instead of indicating which unit calls another, we show which unit exports entities to other units.

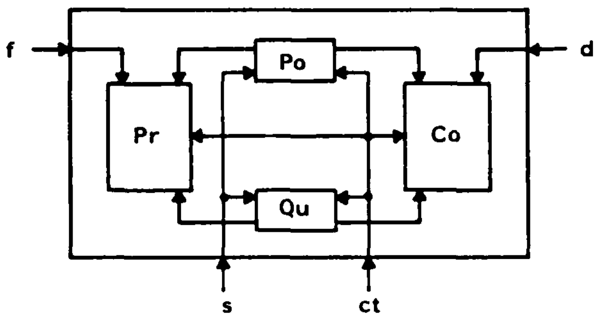
First we focus our attention on the Bucket_Pool module. This module encapsulates a pool of buckets. The size of this pool will remain fixed for one particular application. However, for more reusability the size should be imported from outside. This will allow one to generate different versions of the Bucket_Pool that differ in their sizes.

Next we note that the logical soundness of this design does not depend on any particular representation of the Buckets in the Pool, only the efficiency of the overall system depends on it. Thus in order to enhance the reusability of the Bucket_Pool module one must import the representation (or type) of the Buckets from outside. This extra argument will allow one to produce Pools of various sizes and types from a single implementation alone.

Similar design decisions are valid for the Bucket_Queue module too. That is, it too imports the Queue_Size and the Bucket_Type from some external module. This decision gives us another advantage: the representation of the Buckets can be tested and verified independently of the context in which they are used. At this point we do not anticipate any change in the entities exported from these two modules.

Furthermore, the producer and consumer modules import the operations for filling and draining out the buckets. Since these operations may be specific to the particular application, we abstract out these dependencies and provide a further capability to change these aspects. Thus the module interconnection diagram looks as follows:

T : Bucket_Transfer



s	:	Buffer_Size
ct	:	Bucket_Type
f	:	Fill_Bucket
d	:	Drain_Bucket

Po	:	Bucket_Pool
Qu	:	Bucket_Queue
Pr	:	Bucket_Producer
Co	:	Bucket_Consumer

Figure 10.3 -- Bucket Transfer Module

Next we consider the development of the Bucket_Pool package. This module provides two orthogonal functionalities: a monitor that encapsulates and protects a Pool object, and general purpose operations for manipulating the Pool. Therefore the first design decision we make is to implement these functionalities separately. The major benefit will be the ability to use these functions in other domains as well. However, with respect to reusability the initial requirements are somewhat incomplete: we are not required to check if the Pool is full before returning a bucket back to the Pool. Thus we would like to have a more complete abstraction that can be used in other circumstances.

Thus the monitor module can be designed to import the following entities: the representations of Bucket_Type, Pool_Type, tests for determining the overflow and underflow conditions, and operations for inserting and removing bucket elements from arbitrary pools. It exports the operations for allocating and deallocating elements from a pool.

We also need a complementary module that provides the appropriate operations and representations of Pool objects. We call this module the Pool_Ops module. This module imports the representation of Bucket_Type, and the size of Pool required. In return it exports the implementation of Pool_Type, and the corresponding operations and tests for objects of Pool_Type and Bucket_Type. The resulting module interconnection diagram is shown below.

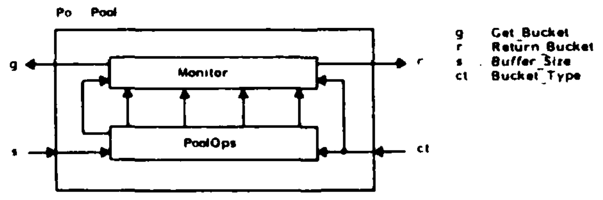


Figure 10.4. -- Bucket_Pool Module

Exactly the same considerations apply to the design of the Bucket_Queue module. However, we note an immediate pay-off of an earlier design decision. The decision to separate and complete the Monitor functionality allows us to retain the same monitor module as before. In this case it must be connected to an appropriate Queue_Ops module for providing mutually exclusive operations on Queue objects rather than Pool objects. Thus the module interconnections for the Bucket_Queue appear as follows.

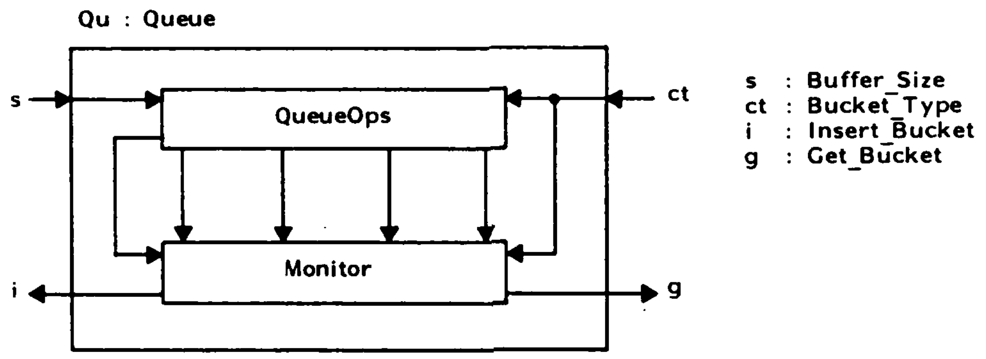


Figure 10.5 -- Bucket_Queue Module

Thus, apart from describing the implementations of the various modules, our design is more or less complete. The details are described in Appendix I.

10.5 SUMMARY

In conclusion we describe another application to which this design can be applied very easily. This application is a conventional line-printer spooler. Instead of transferring buckets we are interested in transferring files to a line-printer driver process. This means that an appropriate representation of files could be substituted for the bucket in this design. The operations of filling and emptying buckets are replaced by the operations of fetching files to be printed and the low-level operations for printing files on the printer.

The problem we have addressed in this report is by no means typical of hard real-time problems. On the contrary it is a soft real-time problem for which typical real-time design techniques, such as cyclical executives, are commonly applied. Our design technique addresses the reusability for such problems in a natural and convenient way. The conditions for reuse (or compatibility) between modules are purely syntactic (or structural). In other words, a given module may be shown to be reusable in another module or application if it is at least structurally compatible. However this is not a sufficient condition for reusability. Nevertheless, it is a start in the right direction.

A much stronger condition for reuse of a module must be syntactic as well as semantic. By this we mean, that the module to be reused must fit into the syntactic structure of the reuser, but also the condition or assertions of the reused module should be the same or equivalent to those required in the reusing context. In hard real-time systems, we can generalize by saying that the semantic structure of the modules must include both timing and functional characteristics of the module. What we require is a facility for stating the structural as well as functional and timing characteristics of modules and for using that as a basis for correct module interconnection criterion. We would like to consider these as subjects for future research.

CHAPTER 11

REFERENCES

[1] US DoD, Military Standard: Ada Programming Language, ANSI/MIL-STD-1815A, January 1983.

[2] Abrams, M. Singhal, S.K. Tripathi, and A.K. Agrawala, Modeling Approaches to Parallel Software Running on Dedicated Processors, Technical Report TR-1296, System Design and Analysis Group, Department of Computer Science, University of Maryland, 1983

[3] Alford, M. W., "A Requirements Engineering Methodology for Real-Time Processing Requirements," IEEE Transactions in Software Engineering, Vol. SE-3, No. 1, January 1977

[4] Anderson, T., Knight, J.C., Practical Software Fault Tolerance for Real-Time Systems, Technical Report TRS 169, University of Newcastle upon Tyne, Computing Laboratory, England, June 1981.

[5] Braun, C., J.B. Goodenough, R.S. Eanes, Ada Reusability Guidelines, SofTech Technical Report 3285-2-208/2, April 1985.

[6] Boeing AIMS Phase 1 Report, Boeing Corp, November 1984

[7] Booch, G., Software Engineering in Ada, Benjamin/Cummings Publishing Co. 1983

[8] Cardelli, L., R. Pike, "Squeak: A Language for Communicating with Mice," ACM SIGGRAPH Conference Proceedings, San Fransisco, Jul 1985.

[9] Cohen, R.M., "Formal Specifications for Real-Time Systems," Proceedings of the IEEE Seventh Texas Conference on Computing Systems, October 1978

[10] Coolahan Jr.,J.E., N. Roussopoulos, "Timing Requirements for Time Driven Systems Using Augmented Petri Nets," IEEE transactions on Software Engineering, Vol. SE-9, No. 5, September 1983

REFERENCES

[11] Dubery, J.M., and A.J. Pinches, "Software for an Air Pollution Measuring System: an Application of Modula," Software-Practice and Experience, Vol. 15, No. 4, pp 413-422, John Wiley, April 1985.

[12] Duncan, A.G. and J.S. Hutchinson, "Using Ada for Industrial Embedded Microprocessor Applications," SIGPLAN Notices, November 1980

[13] Faulk, S. R. and D. L. Parnas, "On the Uses of Synchronization in Hard-Real-Time Systems," Proceedings of the Real-Time Systems Symposium, December 1983, Computer Society Press

[14] Glass, R., "Real Time: The 'Lost World' of Software Debugging and Testing" ACM Communications, Vol. 23, No. 5, May 1980

[15] Gligor, V.D., and G.L. Luckenbaugh, "An Assessment of the Real-Time Requirements for Programming Environments and Languages," Proceedings of the Real-Time Systems Symposium, December 1983, Computer Society Press

[16] Gonzalez, M.J., Jr., "Deterministic Processor Scheduling," ACM Computing Surveys, Vol. 9., No. 3., September 1977.

[17] Greene, W.R., A Guide for the use of Fast Interrupt Entries, KIT Ada-1106-39-CG/CO/MI/RT, SofTech, Inc, Waltham, MA, June 1985.

[18] Grover, V., On Expresssing Module Interconnections in Ada, SofTech Technical Report TP 208, July 1985.

[19] Haase, V., "Real Time Behavior of Programs," IEEE Transactions on Software Engineering, Vol. se-7, No. 5, September 1981

[20] Habermann, A.N., and I.R. Nassi, Efficient Implementation of Ada Tasks, Technical Report CMU-CS-81-147, Carnegie-Mellon University, Pittsburg, PA, June 1981.

[21] Hall, D.L, J.J. Gibbons, and D.A. Woodle, "Avoid Disaster: The use of an Integrated Tool for Managing Throughput and Response Time in Embedded Real-Time Software Systems," Conference on Software Tools, April 1985

[22] Heidelberger, P., and K.S. Trivedi, "Queueing Networks for Parallel Processing with Asynchronous Tasks," IEEE Transactions on Computers, Vol. C-31, No. 11, November 1982

[23] Heidelberger, P., and K.S. Trivedi, "Analytic Queueing Models for Programs with Internal Concurrency," IEEE Transactions on Computers, Vol. C-32, No. 1, January 1983

REFERENCES

[24] Hilfinger, P.N., "Implementation Strategies for Ada Tasking Idioms," Proceedings of the AdaTec Conference on Ada, October, 1982.

[25] Lazowska, E. D., J. Zahorjan, G. S. Graham, and K. C. Sevcik, Quantitative System Performance, Prentice Hall, 1984

[26] MacLaren "Evolving toward Ada in Real Time Systems," SIGPLAN Notices, 1980

[27] McBride,J., Graph Analysis and Design Technique Methodology, Softech, Sept 13, 1984

[28] Mekly, L. J., and S. S. Yau, "Software Design Representation Using Abstract Process Networks," IEEE Transactions on Software Engineering, Vol. SE-6, No. 5, September 1980

[29] Mok, A.K., and M.L. Dertouzos, "Microprocessor Scheduling in a Hard Real-Time Environment," Proceedings of the IEEE Seventh Texas Conference on Computing Systems, October 1978

[30] Mandrioli, D., R. Zicari, C. Ghezzi, F. Tisato, "Modeling the Ada Task System by Petri Nets," Computer Languages, Vol. 10, No. 1, 1985

[31] Molloy, M. K., "Discrete Time Stochastic Petri Nets," IEEE Transactions on Software Engineering," Vol. SE-11, No. 4, April 1985

[32] Nelson, R. A., L. M. Haibt, P. B. Sheridan, "Casting Petri Nets into Programs," IEEE transactions on Software Engineering, Vol. SE-9, No. 5, September 1983

[33] Noe, J. D., and G. J. Nutt, "Macro E-Nets for Representation of Parallel Systems," IEEE Transactions on Software Engineering, Vol. C-22, No. 8, August 1973

[34] Peterson, J. L., Petri Net Theory and the Modeling of Systems, Prentice Hall, 1981

[35] Phillips and Stevenson, "The Role of Ada in Real-Time Embedded Applications," Ada Letters, Vol. 3 No. 4 January 1984

[36] Polise, A.G., and G. Moskovitz, "A Real Time Application of Microprocessors to a Radar System," IEEE Southeastcon, April 1981

[37] Rajeev, S., Certain Optimizations in Ada Tasking Implementations Technical Report 9074-2, SofTech, Waltham, MA, January 1983.

[38] Rajeev, S., On Applying Ada to Real-Time Systems: The Inversion Technique and Some Examples, Technical Report TP 148, SofTech, Inc., Waltham, MA, March 1983.

[39] Ramamoorthy, C.V., and G. S. Ho, "Performance Evaluation of Asynchronous Concurrent Systems Using Petri Nets," IEEE Transactions on Software Engineering, Vol. SE-6, No. 5, September 1980

[40] Randell, B., Lee, P.A., Treleaven, P.C., "Reliability Issues in Computing System Design," ACM Computing Surveys, Vol. 10 no. 2, June 1978.

[41] Roberts, E.A., A. Evans, C.R. Morgan, E.M. Clarke, "Task Management in Ada - A Critical Evaluation for Real-Time Multiprocessors," Software-Practice and Experience, Vol. 11 No. 10, pp. 1019-1051, 1981.

[42] Roman, G.C., "A Taxonomy of Current Issues in Requirements Engineering," Computer (IEEE), April 1985

[43] Ross, D.T., "Applications and Extensions of SADT," Computer (IEEE), April 1985

[44] Sholl, H.A., and T. L. Booth, "Software Performance Modeling Using Computation Structures," IEEE Transactions on Software Engineering, Vol. SE-1, No. 4, December 1975.

[45] SofTech, Inc., Final Technical Report: ALS Optimization Study, Technical Report 1106-17, SofTech, Inc., Waltham, MA, July 1984.

[46] Ward, S.A., "An Approach to Real-Time Computation," Proceedings of the IEEE Seventh Texas Conference on Computing Systems, Oct 1978

[47] Wirth, N. "Towards a Discipline of Real Time Programming," ACM Communications, Vol. 20 No. 8, August 1977

APPENDIX A

WHAT TO DO IN CASE OF A TIMING FAULT

Cyclic Executive Systems

Problem Indicator	Action	
Frame Overrun	Run time	- Take action to desensitize the system to timing failure.
	Offline	- Retune system to eliminate the possibility of frame overrun.
Overloaded Queue	Run time	- Manipulate hardware to provide less input data.
		- Introduce tests to discard less important input data.
		- Reallocate time in cycle so that the software that empties the queue has more time to run.
	Offline	- Improve the efficiency of the routines that empty the queue.
		- Retune the system so the software that empties the queue is given more time.
		- Buy a faster processor.

WHAT TO DO IN CASE OF A TIMING FAULT

Missed Deadline	Run time	- Carry out application dependent algorithm to avoid damage to system hardware and recover from failure if possible.
	Offline	- Retune system to eliminate the possibility of this missed deadline.
		- Improve the efficiency of the affected algorithms.

WHAT TO DO IN CASE OF A TIMING FAULT

Data Driven Systems

Problem Indicator	Action	
Overloaded Queue or Low Priority Tasks not Running	Run time	- Manipulate hardware to provide less input data.
		- Introduce tests to discard less important input data.
	Offline	- Improve the efficiency of the routines that empty the queue.
		- Raise the priority of the routine that empties the queue.
		- Buy a faster processor.
Missed Deadline	Run time	- Carry out application dependent algorithm to avoid damage to system hardware and recover from failure if possible.
	Offline	- Use a tuning strategy to eliminate the possibility of this missed deadline. Possible strategies include manipulating the schedule of input data arrival, or replacing competing tasks with co-routines.
		- Improve the efficiency of the affected algorithms.

APPENDIX B

CODING EXAMPLES

This case shows software that adapts itself to the current mode.

Cyclic Executive case --

```
procedure Routine is
begin
  .
  .
  case Mode is
    when Mode_1 =>
      .
      .
    when Mode_2 =>
      .
      .
  end case;
  .
  .
end Routine;
```

CODING EXAMPLES

Data Driven Design case --

```
task body Routine is
begin
  loop
    .
    .
    case Mode is
      when Mode_1 =>
        .
        .
      when Mode_2 =>
        .
        .
    end case;
    .
    .
  end loop;
end Routine;
```

A cyclic executive system may change its frame assignments

```
task body Cyclic_Executive is
begin
  loop
    case Mode is
      when Mode_1 =>
        accept Tick;
          Routine_1;
          Routine_2;
        accept Tick;
          Routine_3;
          Routine_4;
          Routine_5;
      when Mode_2 =>
        accept Tick;
          Routine_1;
          Routine_2;
        accept Tick;
          Routine_1;
          Routine_2;
          Routine_5;
    end case;
  end loop;
end Cyclic_Executive;
```

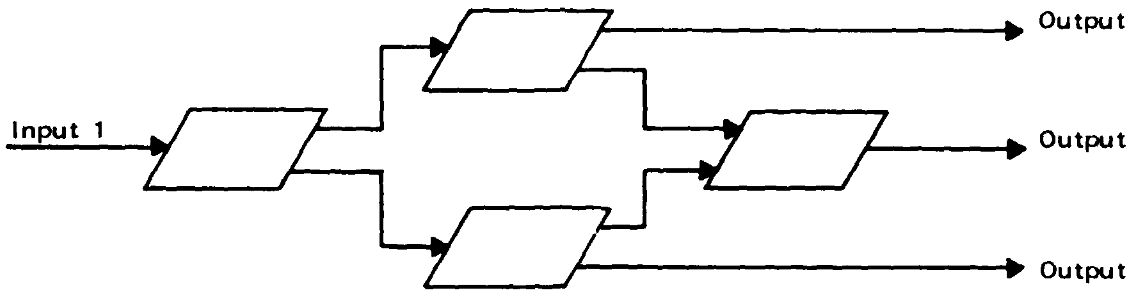
Data Driven Designs can adjust mode by changing data flow --

```
task body Producer is
begin
  loop
    .
    .
    case Mode is
      when Mode_1 =>
        Buffer_1.Put;
      when Mode_2 =>
        Buffer_2.Put;
    end case;
    .
    .
  end loop;
end Producer;
```

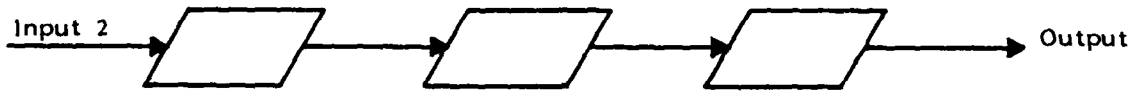
```
Cyclic Executives can completely redefine the cyclic structure --
task body Cyclic_Executive is
begin
  loop
    case Mode is
      when Mode_1 =>
        accept Tick;
          Routine_1;
          Routine_2;
        accept Tick;
          Routine_3;
          Routine_4;
          Routine_5;
      when Mode_2 =>
        accept Tick;
          Routine_1;
          Routine_2;
        accept Tick;
          Routine_1;
          Routine_2;
          Routine_3;
        accept Tick;
          Routine_1;
          Routine_2;
          Routine_4;
        accept Tick;
          Routine_1;
          Routine_2;
          Routine_5;
      when Mode_3 =>
        accept Tick;
          Routine_1;
          Routine_2;
    end case;
  end loop;
end Cyclic_Executive;
```

Data Driven designs can have multiple independent structures --

Multiple Independent Data Flows



Mode 1



Mode 2

APPENDIX C

SUGGESTIONS FOR ACHIEVING MODE CHANGING REQUIREMENTS

Data Driven Design

PROBLEM	APPROACH
Data must be vectored from routines in the old mode to routines in the new mode.	Buffer routines can be used to select from multiple inputs and outputs based on mode to provide the proper data flow for the current mode.
Inputs not used in the new mode must not be allowed to enter the data flow.	Buffer tasks can reject inputs based on the system mode.
Data left over from the previous mode causes unnecessary routines to continue running in the new mode.	Buffer tasks can flush their queues of data on certain mode changes.
Mode information must be controlled to protect the integrity of the mode data.	Mode can be controlled by a single package with reads and writes controlled so that a write cannot overlap a read or another write.
Mode information must be disseminated in such a way that it arrives at affected tasks in a specific order.	Mode can be explicitly distributed by a mode distribution task. or Mode can be attached to a data structure that flows through all the affected routines.

SUGGESTIONS FOR ACHIEVING MODE CHANGING REQUIREMENTS
Data Driven Design

PROBLEM	APPROACH
Two modes must be mutually exclusive.	Introduce a task between the modes to control the transfer. or Encapsulate critical mode sections in the same task.
Software from different modes must be overlayed.	A task that manages the overlay memory space can be introduced. This problem is similar to virtual memory management.
Time of mode change is subject to deadline verification.	Introduce a task between the modes to control the transfer. or Encapsulate critical mode sections in the same task.

Cyclic Executive

PROBLEM	APPROACH
Data must be vectored from routines in the old mode to routines in the new mode.	Routines can be modified to pick-up or place data in different places based on mode. or New routines can be hard coded as to where the data they want is located.
Inputs not used in the new mode must not be allowed to enter the data flow.	Routines will not be scheduled to read the data in.
Data left over from the previous mode is lost.	Routines can be designed to evaluate left over data picking out anything useful and discarding the rest.
Software from different modes must be overlayed.	A pause between modes is introduced to load the new software. or A transitional mode can be introduced which runs a high priority nucleus of the new mode while the rest of the mode is loaded.
Mode must be changed for processor resource reallocation.	Frame assignments can be replaced. or A new cyclic structure can be introduced.

APPENDIX D

EVALUATION TABLE OF ANALYSIS ALTERNATIVES

EVALUATION TABLE OF ANALYSIS ALTERNATIVES
Queueing Networks

Measurement	Verification	Numeric Estimate	Problem Location
Maximum Response Time	N	N	N
Average Response Time	S	S	N
Average Jitter	N	N	N
Maximum Jitter	N	N	N
Throughput	S	S	S
Processor Utilization	S	S	S

Pros - 1) Can be used very early in system design.
2) Models key elements of the hardware architecture.
3) Can produce important estimates with very little information.

Cons - 1) All estimates are statistical in nature.
2) A limited number of measurement types can be estimated.
3) Some systems are difficult to represent as a queueing model.
4) Significant designer expertise is required to develop these models.
5) These models cannot be decomposed to represent more detailed software designs.

S = Statistical Measurement Possible
D = Deterministic Measurement Possible
E = Either Possible
N = Neither Possible

EVALUATION TABLE OF ANALYSIS ALTERNATIVES
Timed Petri Net Modeling

Measurement	Verification	Numeric Estimate	Problem Location
Maximum Response Time	D	D	E
Average Response Time	E	E	E
Average Jitter	E	E	E
Maximum Jitter	D	D	E
Throughput	E	E	E
Processor Utilization	E	E	E

Pros - 1) Can represent a wide variety of software systems.
2) Can support either a statistical or deterministic representation of time consumption.

Cons - 1) There is no clear correlation between Petri net elements and software components (a place does not have a consistent meaning).
2) There is no right or wrong way to model software (there might be a dozen different nets that correctly model the same piece of software).
3) These nets contain much unneeded flexibility which complicates analysis.
4) The nets become complex and hard to follow in large systems.

S = Statistical Measurement Possible
D = Deterministic Measurement Possible
E = Either Possible
N = Neither Possible

EVALUATION TABLE OF ANALYSIS ALTERNATIVES
AP Network modeling

Measurement	Verification	Numeric Estimate	Problem Location
Maximum Response Time	D	D	E
Average Response Time	E	E	E
Average Jitter	E	E	E
Maximum Jitter	D	D	E
Throughput	E	E	E
Processor Utilization	E	E	E

Pros - 1) There is a single correct way to model software.
2) Flexibility is limited by directing the modeling towards structured software.
3) Data flow can be differentiated from the flow of control where necessary.
4) The network components are easily decomposed to form a more detailed design.
5) These networks are easy to build and easy to understand.

Cons - 1) Interprocess communication is not well defined in the model.

S = Statistical Measurement Possible
D = Deterministic Measurement Possible
E = Either Possible
N = Neither Possible

EVALUATION TABLE OF ANALYSIS ALTERNATIVES
Markov Chain Modeling

Measurement	Verification	Numeric Estimate	Problem Location
Maximum Response Time	N	N	N
Average Response Time	S	S	S
Average Jitter	S	S	S
Maximum Jitter	N	N	N
Throughput	S	S	S
Processor Utilization	S	S	S

Pros -
1) These chains are as powerful a model as stochastic Petri nets.
2) The Markov chain representation can be simplified for easier understanding and analysis.

Cons -
1) The simplification of the chain can significantly increase the complexity of the net.
2) These models can only represent statistical timing data. This means some important concepts cannot be considered.

S = Statistical Measurement Possible
D = Deterministic Measurement Possible
E = Either Possible
N = Neither Possible

EVALUATION TABLE OF ANALYSIS ALTERNATIVES
Finite State Automata

Measurement	Verification	Numeric Estimate	Problem Location
Maximum Response Time	D	D	E
Average Response Time	E	E	E
Average Jitter	E	E	E
Maximum Jitter	D	D	E
Throughput	E	E	E
Processor Utilization	E	E	E

Pros -
1) These automata provide enough power to model the performance of most software.
2) These automata are easier to analyze than Petri nets.

Cons -
1) This technique is almost identical with Petri net modeling, only with less capability.

S = Statistical Measurement Possible
D = Deterministic Measurement Possible
E = Either Possible
N = Neither Possible

EVALUATION TABLE OF ANALYSIS ALTERNATIVES
Computation Structures

Measurement	Verification	Numeric Estimate	Problem Location
Maximum Response Time	D	D	E
Average Response Time	E	E	E
Average Jitter	E	E	E
Maximum Jitter	D	D	E
Throughput	E	E	E
Processor Utilization	E	E	E

Pros - 1) This technique maintains a clear distinction between flow of control and data flow.
2) This technique has as much power as Petri net modeling.

Cons - 1) The complexities of Petri net modeling are further compounded by having to deal with two interrelated graphs.

S = Statistical Measurement Possible
D = Deterministic Measurement Possible
E = Either Possible
N = Neither Possible

APPENDIX E

INTERFACE PROCESSING

Event counts are read from addresses 8#100204# thru 8#100216#. The event counts should be read as the event gate is closed. The Contact Data FIFO utilizes two addresses for control. Location 8#100220# is the active word count. This indicates how many words are currently in the FIFO. Location 8#100221# is the input address. By reading this address, the program retrieves the next word from the FIFO. The FIFO should be emptied at least once every 1.5 degrees (20.8 msec) while a contact gate is active. The FIR interface uses three locations for passing control and data information. Location 8#100200# is used as the data input/output location. Location 8#100201# is used as a usage indicator. If it is set to one, the data location is communicating with the FIR command register. If it is set to two the data location is communicating with FIR memory. Location 8#100202# is used as a trigger to cause data transfers. In order to access FIR memory the following steps are necessary. The FIR command register must be placed in 'set address' mode, the FIR memory address desired must be sent, the FIR control register must be set in 'read word' mode or 'read series' mode, and the desired data must be retrieved. Commands to the FIR control register are defined as follows:

0 =	Test 1
1 =	Test 2
2 =	Load Address
4 =	Single Word Access Mode
8 =	Word Series Access Mode

An Example FIR Interface Handling Package.

```
package Low_Level is
  procedure Read_Location  (Address: in Data_Type; Data: out Data_Type);
  procedure Read_Series    (Address: in Data_Type; Data: out Data_Array_Type);
  procedure Write_Location (Address: in Data_Type; Data: in Data_Type);
  procedure Write_Series   (Address: in Data_Type; Data: in Data_Array_Type);
end Low_Level;

package body Low_Level is

  -- Control Registers
  Data_Location    : Data_Type;
  for Data_Location    use at 8#100200#;
  Command_Location: Data_Type;
  for Command_Location use at 8#100201#;
  Strobe_Location : Data_Type;
  for Strobe_Location  use at 8#100202#;

  -- Command Register Options
  Command  : constant Data_Type := 1;
  Interface: constant Data_Type := 2;

  -- Command Options
  Test_1      : constant Data_Type := 0;
  Test_2      : constant Data_Type := 1;
  Load_Address: constant Data_Type := 2;
  Single_Word : constant Data_Type := 4;
  Word_Series : constant Data_Type := 8;

  -- Strobe Value
  Strobe: constant Data_Type := 0;

  -- Location to put dummy reads
  Garbage: Data_Type;
  pragma shared (Garbage);
```

```
-- The following procedure reads a location of FIR memory

procedure Read_Location (Address: in Data_Type; Data: out Data_Type) is
begin
  Command_Location := Command;          -- Set Command Mode
  Data_Location := Load_Address;        -- Set Load Address Command
  Strobe_Location := Strobe;            -- Strobe Command into Interface
  Command_Location := Interface;        -- Set Interface Mode
  Data_Location := Address;             -- Place Address in Interface
  Strobe_Location := Strobe;            -- Strobe Address into Interface
  Command_Location := Command;          -- Set Command Mode
  Data_Location := Single_Word;         -- Set Single Word Transfer Command
  Strobe_Location := Strobe;            -- Strobe Command into Interface
  Command_Location := Interface;        -- Set Interface Mode
  Garbage := Data_Location;             -- Put Interface into Read Direction
  Strobe_Location := Strobe;            -- Strobe Data into Interface
  Data := Data_Location;                -- Read Data
end Read_Location;
```

```
-- The following routine Reads a series of 100 FIR locations.

procedure Read_Series (Address: in Data_Type; Data: out Data_Array_Type) is

  i: integer;

begin
  Command_Location := Command;        -- Set Command Mode
  Data_Location := Load_Address;      -- Set Load Address Command
  Strobe_Location := Strobe;          -- Strobe Command into Interface
  Command_Location := Interface;      -- Set Interface Mode
  Data_Location := Address;           -- Place Address in Interface
  Strobe_Location := Strobe;          -- Strobe Address into Interface
  Command_Location := Command;        -- Set Command Mode
  Data_Location := Word_Series;       -- Set Word Series Transfer Command
  Strobe_Location := Strobe;          -- Strobe Command into Interface
  Command_Location := Interface;      -- Set Interface Mode
  Garbage := Data_Location;           -- Put Interface into Read Direction
  for i := 1 to 100 loop
    Strobe_Location := Strobe;        -- Strobe Data into Interface
    Data(i) := Data_Location;         -- Read Data
  end loop;
end Read_Series;
```

```
-- The following routine writes a single word to FIR memory.

procedure Write_Location (Address: in Data_Type; Data: in Data_Type) is
begin
  Command_Location := Command;         -- Set Command Mode
  Data_Location := Load_Address;       -- Set Load Address Command
  Strobe_Location := Strobe;           -- Strobe Command into Interface
  Command_Location := Interface;       -- Set Interface Mode
  Data_Location := Address;            -- Place Address in Interface
  Strobe_Location := Strobe;           -- Strobe Address into Interface
  Command_Location := Command;         -- Set Command Mode
  Data_Location := Single_Word;        -- Set Single Word Transfer Command
  Strobe_Location := Strobe;           -- Strobe Command into Interface
  Command_Location := Interface;       -- Set Interface Mode
  Data_Location := Data;               -- Place Data in interface
  Strobe_Location := Strobe;           -- Strobe Data into Interface
end Write_Location;
```

```
  -- The following routine writes a 100 word series of data to FIR memory.

  procedure Write_Series (Address: in Data_Type; Data: in Data_Array_Type) is

    i: integer;

  begin
    Command_Location := Command;          -- Set Command Mode
    Data_Location := Load_Address;        -- Set Load Address Command
    Strobe_Location := Strobe;            -- Strobe Command into Interface
    Command_Location := Interface;        -- Set Interface Mode
    Data_Location := Address;             -- Place Address in Interface
    Strobe_Location := Strobe;            -- Strobe Address into Interface
    Command_Location := Command;          -- Set Command Mode
    Data_Location := Word_Series;         -- Set Word Series Transfer Command
    Strobe_Location := Strobe;            -- Strobe Command into Interface
    Command_Location := Interface;        -- Set Interface Mode
    for i := 1 to 100 loop
      Data_Location := Data(i);           -- Place Data in Interface
      Strobe_Location := Strobe;          -- Strobe Data into Interface
    end loop;
  end Write_Series;

end Low_Level
```

APPENDIX F

CYCLIC EXECUTIVE SOLUTION

The Cyclic Executive solution is designed to achieve the following time line:

```
------------     -----------     --------     ----------
|          |     |         |     |      |     |        |
|  Batch   |     |  FIR    |     | Disk |     |Keyboard|
|Processing|     | Handler |     | End  |     |Handler |
|          |     |         |     |      |     |        |
------------     -----------     --------     ----------

Event Driven Processing



T=0 sec                                                                 T=5 sec
|                                                                             |
--------------------------------------------------------------------------------
|          |            |      |        |          |          |      |        |
|Interface |    FIR     | Disk | Target | Contact  |Event     |Tape  |        |
| Testing  | Extraction |Output| Output |Extraction|Extraction|Output|        |
|          |            |      |        |          |          |      |        |
--------------------------------------------------------------------------------

Cyclic Scheduling (Minor Cycle = Major Cycle)



-------------------------------------------
|                     |                   |
| Command Processing  | Display Processing|
|                     |                   |
|                     |                   |
-------------------------------------------

Background Process
```

CYCLIC EXECUTIVE SOLUTION
Task Declarations

```
package Event_Driven_Package is             -- Package Containing High Priority
                                            -- Event Driven Processing

  task Batch_Handler is                     -- a batch occurs approximately every
    entry Batch_Interrupt;                  -- 20.8 msec (or every 1.5 degrees)
    for Batch_Interrupt use at 8#100301#;
    pragma priority (10);
  end Batch_Handler;

  task FIR_Handler is                       -- FIR interrupts occur when FIR data
    entry FIR_Interrupt;                    -- is ready to be extracted.
    for FIR_Interrupt use at 8#100302#;
    pragma priority (9);
  end FIR_Handler;

  task Disk_End is                          -- This interrupt occurs when a DMA
    Entry Disk_Interrupt;                   -- transfer completes.
    for Disk_Interrupt use at 8#100303#;
    pragma priority (8);
  end Disk_End;

  task Keyboard_Handler is                  -- This interrupt occurs whenever a
    Entry New_Character;                    -- key is pressed on the keyboard.
    for New_Character use at 8#100304#;
    pragma priority (4);
  end Keyboard_Handler;

end Event_Driven_Package;


package Cyclic_Routines is                  -- Package of all processing that
                                            -- runs at the standard cyclic rate
  task Cyclic_Executive is
    entry Minor_Cycle_Start;
    for Minor_Cycle_Start use at 8#100300#;
    pragma priority (5);
  end Cyclic_Executive;

end Cyclic_Routines;


package Background_Routines is              -- Package of all processing that
                                            -- runs in background.
  task Background_Task is
   pragma priority (2);
  end Background_Task;

end Background_Routines;
```

```
package Data_Definitions is             -- This Package Defines the various
                                        -- data types to be used in this
                                        -- system.

  type Data_Word_Type is array(1..16) of Boolean;
  pragma pack (Data_Word_Type);
  type Data_Holder_Type is array(Integer range <>) of Data_Word_Type;
  subtype Data_Array_Type is Data_Holder_Type(1..32767);
  subtype Buffer_Number_Type is Integer range 0..3;
  subtype Control_Words_Type is Data_Holder_Type(1..5);
  subtype Contact_Buffer_Type is Data_Holder_Type(1..2*11);
  subtype Small_Contact_Type is Data_Holder_Type(1..42);

  subtype Event_Buffer_Type is Data_Holder_Type(1..10);

  type Tape_Mode_Type is (Events, Contacts, Both);

  type Slant_Ranges is array(1..2) of Float range 0.0..Float'Last;

  subtype Bearing_Type is Float range 0.0..360.0;

  type Bearings is array(1..2) of Bearing_Type;

  type Gate_Type is
    record
      Active             : Boolean := False;
      On                 : Boolean := False;
      Range_Parameters   : Slant_Ranges;
      Bearing_Parameters : Bearings;
    end record;

  type Selection_Type is (Target, Clutter);

  type Target_Type is
    record
      Active               : Boolean := False;
      On                   : Boolean := False;
      Selection            : Selection_Type;
      Range_Parameters     : Slant_Ranges;
      Range_Rate           : Float;
      Range_Acceleration   : Float;
      Bearing_Parameters   : Bearings;
      Bearing_Rate         : Float;
      Bearing_Acceleration : Float;
    end record;

  type Target_Pair_Type is array(1..2) of Target_Type;

  subtype Command_String is String(1..40);
end Data_Definitions;
```

CYCLIC EXECUTIVE SOLUTION
Shared Data to Background

```
with Data_Definitions;
use Data_Definitions;
package Data_to_Background is

  -- FIR Data to Background -
  -- When New_FIR_Data is set, background should pickup Control_Words
  -- and reset New_FIR_Data.
  New_FIR_Data  : Boolean := False;
  pragma Shared (New_FIR_Data);
  Control_Words : Control_Words_Type;
  pragma Shared (Control_Words);

  -- Contact Data to Background -
  -- When New_Contacts_to_Background is set, background should pickup
  -- Display_Contacts and reset New_Contacts_to_Background.
  New_Contacts_to_Background : Boolean := False;
  pragma Shared (New_Contacts_to_Background);
  Display_Contacts           : Small_Contact_Type;
  pragma Shared (Display_Contacts);

  -- Event Data to background -
  -- When New_Events_to_Background is set, background should pickup
  -- Display_Events and reset New_Events_to_Background.
  New_Events_to_Background : Boolean := False;
  pragma Shared (New_Events_to_Background);
  Display_Events           : Event_Buffer_Type;
  pragma Shared (Display_Events);

  -- Command to Command Processing
  New_Command : Boolean := False;
  Command     : Command_String;

end Data_to_Background;
```

```
with Data_Definitions;
use Data_Definitions;
package Data_to_Event is

  -- Background data to Event driven -
  -- When New_Background_Data is set, the Event Driven package should
  -- read in the gate parameters and reset New_Background_Data.
  New_Background_Data : Boolean := False;
  pragma Shared (New_Background_Data);
  Contact_Gate        : Gate_Type;
  pragma Shared(Contact_Gate);
  Event_Gate          : Gate_Type;
  pragma Shared(Event_Gate);
  Extraction_Gate     : Gate_Type;
  pragma Shared(Extraction_Gate);

  -- Cyclic Data to Event driven package -
  -- When New_Cyclic_Data is set, the Event driven package should
  -- read in the target parameters and reset New_Cyclic_Data.
  New_Cyclic_Data : Boolean := False;
  pragma Shared(New_Cyclic_Data);
  Targets         : Target_Pair_Type;
  pragma Shared(Targets);

end Data_to_Event;
```

CYCLIC EXECUTIVE SOLUTION
Shared Data to Cyclic Routines

```
with Data_Definitions;
use Data_Definitions;
package Data_to_Cyclic is

  -- Target Input Data -
  -- When New_Data is set, the cyclic package should read
  -- the Target_Input parameters and clear New_Data
  New_Data      : Boolean := False;
  pragma Shared(New_Data);
  Targets_Input : Target_Pair_Type;
  pragma Shared(Targets_Input);

  -- Contact Input Data -
  -- When New_Contacts is set, the cyclic package should read
  -- the Contact_Buffers segment pointed to by Contact_Switch,
  -- and clear New_Contacts.
  New_Contacts    : Boolean := False;
  pragma Shared(New_Contacts);
  Contact_Switch  : Integer range 1..2 := 2;
  pragma Shared(Contact_Switch);
  Contact_Buffers : array(1..2) of Contact_Buffer_Type;
  pragma Shared(Contact_Buffers);

  -- Event Input Data -
  -- When New_Events is set, the cyclic package should read
  -- the Event_Buffers segment pointed to by Event_Switch,
  -- and clear New_Events.
  New_Events    : Boolean := False;
  pragma Shared(New_Events);
  Event_Switch  : Integer range 1..2 := 2;
  pragma Shared(Event_Switch);
  Event_Buffers : array(1..2) of Event_Buffer_Type;
  pragma Shared(Event_Buffers);

  Tape_Mode : Tape_Mode_Type;           -- Indicates what data is to be
  pragma Shared(Tape_Mode);             -- recorded to tape.
  Tape_On   : Boolean := False;         -- Indicates whether data is to be
  pragma Shared(Tape_On);               -- recorded to tape.

  FIR_Ready: Boolean := False;          -- Indicates that new FIR data is
  pragma Shared(FIR_Ready);             -- ready for extraction.

  Disk_On: Boolean := False;            -- Indicates whether data should be
  pragma Shared(Disk_On);               -- recorded to disk.

  Disk_is_Done: Boolean := False;       -- Indicates to cyclic routines that
  pragma Shared(Disk_is_Done);          -- a disk recording is complete.

 end Data_to_Cyclic;
```

```
with Data_Definitions;
use Data_Definitions;
package Low_Level is

  procedure FIR_In(Data_Array : out Data_Array_Type);
  function Interface_Test return Boolean;
  procedure Initiate_Disk_Out(Data_Array : in Data_Array_Type);
  procedure Disk_IO_Complete;
  function Disk_is_not_Active return Boolean;
  function FIFO_Empty return Boolean;
  procedure FIFO_Get(Contact_Buffer: out Contact_Buffer_Type);
  procedure Events_Get(Event_Buffer: out Event_Buffer_Type);
  procedure Record_Events(Event_Buffer: in Event_Buffer_Type);
  procedure Record_Contacts(Contact_Buffer: in Contact_Buffer_Type);
  procedure Record_Both(Contact_Buffer: in Contact_Buffer_Type;
                        Event_Buffer: in Event_Buffer_Type);
  function Tape_Done return Boolean;
  function End_of_Tape return Boolean;
  function Get_Character return Character;
end Low_Level;
```

CYCLIC EXECUTIVE SOLUTION
Low Level I/O Package

```
package body Low_Level is
  procedure FIR_In(Data_Array : out Data_Array_Type) is separate;
  -- reads FIR memory into Data_Array

  function Interface_Test return Boolean is separate;
  -- Tests to see if FIR interface is operating properly

  procedure Initiate_Disk_Out(Data_Array : in Data_Array_Type) is separate;
  -- Starts DMA transfer to Disk

  procedure Disk_IO_Complete is separate;
  -- informs low level that the disk Transfer is complete

  function Disk_is_not_Active return Boolean is separate;
  -- tells caller whether the disk is active

  function FIFO_Empty return Boolean is separate;
  -- tells the caller whether the FIFO is empty

  procedure FIFO_Get(Contact_Buffer: out Contact_Buffer_Type) is separate;
  -- places the data in the FIFO in the designate contact buffer

  procedure Events_Get(Event_Buffer: out Event_Buffer_Type) is separate;
  -- Retrieves events from the event counter locations

  procedure Record_Events(Event_Buffer: in Event_Buffer_Type) is separate;
  -- records event data on tape

  procedure Record_Contacts(Contact_Buffer: in Contact_Buffer_Type) is separate;
  -- records contact data on tape

  procedure Record_Both(Contact_Buffer: in Contact_Buffer_Type;
                        Event_Buffer: in Event_Buffer_Type) is separate;
  -- records event and contact data on tape

  function Tape_Done return Boolean is separate;
  -- tells caller whether the last tape operation is complete

  function End_of_Tape return Boolean is separate;
  -- tells caller whether the end of the tape has been reached

  function Get_Character return Character is separate;
  -- returns a character from the keyboard


end Low_Level;
```

```
with Low_Level;
with Data_to_Event;
with Data_to_Background;
with Data_Definitions;
use Data_Definitions;
with Data_to_Cyclic;
use Data_to_Cyclic;

package body Cyclic_Routines is

Data_Array                : array(0..3) of Data_Array_Type;
Input_Buffer_Number       : Buffer_Number_Type := 0;
Output_Buffer_Number      : Buffer_Number_Type := 0;
Number_of_Buffers_Active : Integer range 0..4 := 0;
FIR_Interface_OK          : Boolean := False;
Local_Targets             : Target_Pair_Type;

----------------------------------------------------------

  procedure Interface_Test;
  procedure FIR_Extraction;
  procedure Disk_Storage_Initiation;
  procedure Target_Output;
  procedure Tape_Output;


  task body Cyclic_Executive is               -- This Executive calls the
  begin                                       -- routines in a minor frame for
    loop  -- forever                          -- each five second interrupt.
      accept Minor_Cycle_Start;
      Interface_Test;
      FIR_Extraction;
      Disk_Storage_Initiation;
      Target_Output;
      Tape_Output;
    end loop;
  end Cyclic_Executive;

  procedure Interface_Test is
  begin
    FIR_Interface_OK := Low_Level.Interface_Test;
  end Interface_Test;

  procedure FIR_Extraction is   -- When FIR data is ready for extraction this
  begin                         -- routine reads the data into memory.
    if FIR_Interface_OK then
      if FIR_Ready then
        if Number_of_Buffers_Active < 4 then
          Low_Level.FIR_In(Data_Array(Input_Buffer_Number));
          -- Read in Data into current input buffer
          Data_to_Background.Control_Words(1..5) :=
                Data_Array(Input_Buffer_Number)(16380..16384);
          -- Put Data in buffer to Display
          Input_Buffer_Number := (Input_Buffer_Number + 1) mod 4;
          -- Move input pointer to next buffer.
          Number_of_Buffers_Active := Number_of_Buffers_Active + 1;
        end if;
      end if;
    end if;
  end FIR_Extraction;

  procedure Disk_Storage_Initiation is  -- When there is FIR data in memory
                                        -- and the disk is not active this
                                        -- routine initiates a DMA transfer
  begin                                 -- to disk.
    if Disk_is_Done then
        Number_of_Buffers_Active := Number_of_Buffers_Active - 1;
        Disk_is_Done := False;
    end if;
    if Number_of_Buffers_Active /= 0 then
      if Low_Level.Disk_is_not_Active then
        if Disk_On then
          Low_Level.Initiate_Disk_Out(Data_Array(Output_Buffer_Number));
          -- Start DMA to Disk
        else
          Number_of_Buffers_Active := Number_of_Buffers_Active - 1;
        end if;
        Output_Buffer_Number := (Output_Buffer_Number + 1) mod 4;
        --move output pointer to next buffer
      end if;
    end if;
  end Disk_Storage_Initiation;

  procedure Target_Output is     -- This routine outputs target parameters each
                                 -- scan to the gate controlling software in the
                                 -- event driven package.  The parameters are
                                 -- output each scan so that scan to scan
  begin                          -- can be introduced as an option.
    if New_Data then
      Local_Targets := Targets_Input;
    end if;
    if not (Data_to_Event.New_Cyclic_Data) then
      if Local_Targets(1).On and (not Data_to_Event.Targets(1).On) then
        Data_to_Event.Targets(1) := Local_Targets(1);
        Data_to_Event.New_Cyclic_Data := True;
      end if;
      if Local_Targets(2).On and (not Data_to_Event.Targets(2).On) then
        Data_to_Event.Targets(2) := Local_Targets(2);
        Data_to_Event.New_Cyclic_Data := True;
      end if;
    end if;
  end Target_Output;

  procedure Tape_Output is          -- When Contact And Event Data has been
                                    -- input to memory by the event driven
                                    -- package, this routine outputs the
  begin                             -- data to tape.
    if (Tape_On and (Low_Level.Tape_Done and
       (not (Low_Level.End_of_Tape)))) then
      case Tape_Mode is
        when Events  =>
          if New_Events then
            Low_Level.Record_Events(Event_Buffers(Event_Switch));
             -- record event to tape
            New_Events := False;
          end if;
        when Contacts  =>
          if New_Contacts then
            Low_Level.Record_Contacts(Contact_Buffers(Contact_Switch));
             -- Record Contacts to Tape
            New_Contacts := False;
          end if;
        when Both  =>
          if New_Contacts and New_Events then
            Low_Level.Record_Both(Contact_Buffers(Contact_Switch),
                                  Event_Buffers(Event_Switch));
             -- Record events and contacts to tape.
            New_Contacts := False;
            New_Events := False;
          end if;
      end case;
    end if;
  end Tape_Output;

end Cyclic_Routines;
```

CYCLIC EXECUTIVE SOLUTION
Background Package

```
with Data_to_Cyclic;
with Data_to_Event;
with Data_Definitions;
use Data_Definitions;
with Data_to_Background;
use Data_to_Background;

package body Background_Routines is  -- The routines in this package are run
                                     -- whenever nothing else is running.

  New_Display: Boolean := False;
  Display_Number: Integer;


  procedure Command_Processing;
  procedure Display_Processing;

  task body Background_Task is
  begin
    loop  -- forever
      Command_Processing;
      Display_Processing;
    end loop;
  end Background_Task;


  procedure Command_Processing is      -- This routine receives a command word
                                       -- as input from the keyboard handler in
                                       -- the event driven package.  This
                                       -- command is interpreted and
                                       -- instructions are sent to the rest of
                                       -- of the software in the system to
                                       -- effect the desired result.
    New_Target            : Boolean := False;
    Local_Targets         : Target_Pair_Type;
    New_Gate              : Boolean := False;
    Local_Contact_Gate    : Gate_Type;
    Local_Event_Gate      : Gate_Type;
    Local_Extraction_Gate : Gate_Type;
  begin
    if New_Command then
      -- process command, if it is manipulating gate or target parameters
      -- for future gate output these changes are recorded in the local gate
      -- and target files.  Only one gate or target may be in the process of
      -- modification at any one time.  When the command completes the
      -- modification of a gate the new gate flag will be set.  When the
      -- command completes the modification of a target the new target flag
      -- will be set.  Changes to the display are commanded through the use
      -- of the new display and display number parameters.  If the state of the
      -- disk is to be changed it will be indicated by changing
      -- Data_to_Cyclic.Disk_On.
      if New_Target then
        if not (Data_to_Cyclic.New_Data) then
          Data_to_Cyclic.Targets_Input := Local_Targets;
          Data_to_Cyclic.New_Data := True;
        end if;
      end if;
      if New_Gate then
        if not (Data_to_Event.New_Background_Data) then
          Data_to_Event.Contact_Gate := Local_Contact_Gate;
          Data_to_Event.Event_Gate := Local_Event_Gate;
          Data_to_Event.Extraction_Gate := Local_Extraction_Gate;
          Data_to_Event.New_Background_Data := True;
        end if;
      end if;
    end if;
  end Command_Processing;

  procedure Display_Processing is      -- This routine is capable of maintaining
                                       -- several status and information
                                       -- displays on the system console.  It
                                       -- receives commands from command
                                       -- processing and displays the
                                       -- appropriate data collected from the
                                       -- rest of the system.
    Local_Display_Number: Integer;
  begin
    if New_Display then
      Local_Display_Number := Display_Number;
    end if;
     -- update display information
    New_FIR_Data := False;              -- ask for new data from inputs
    New_Contacts_to_Background := False;
    New_Events_to_Background := False;
  end Display_Processing;

end Background_Routines;
```

CYCLIC EXECUTIVE SOLUTION
Event Driven Package

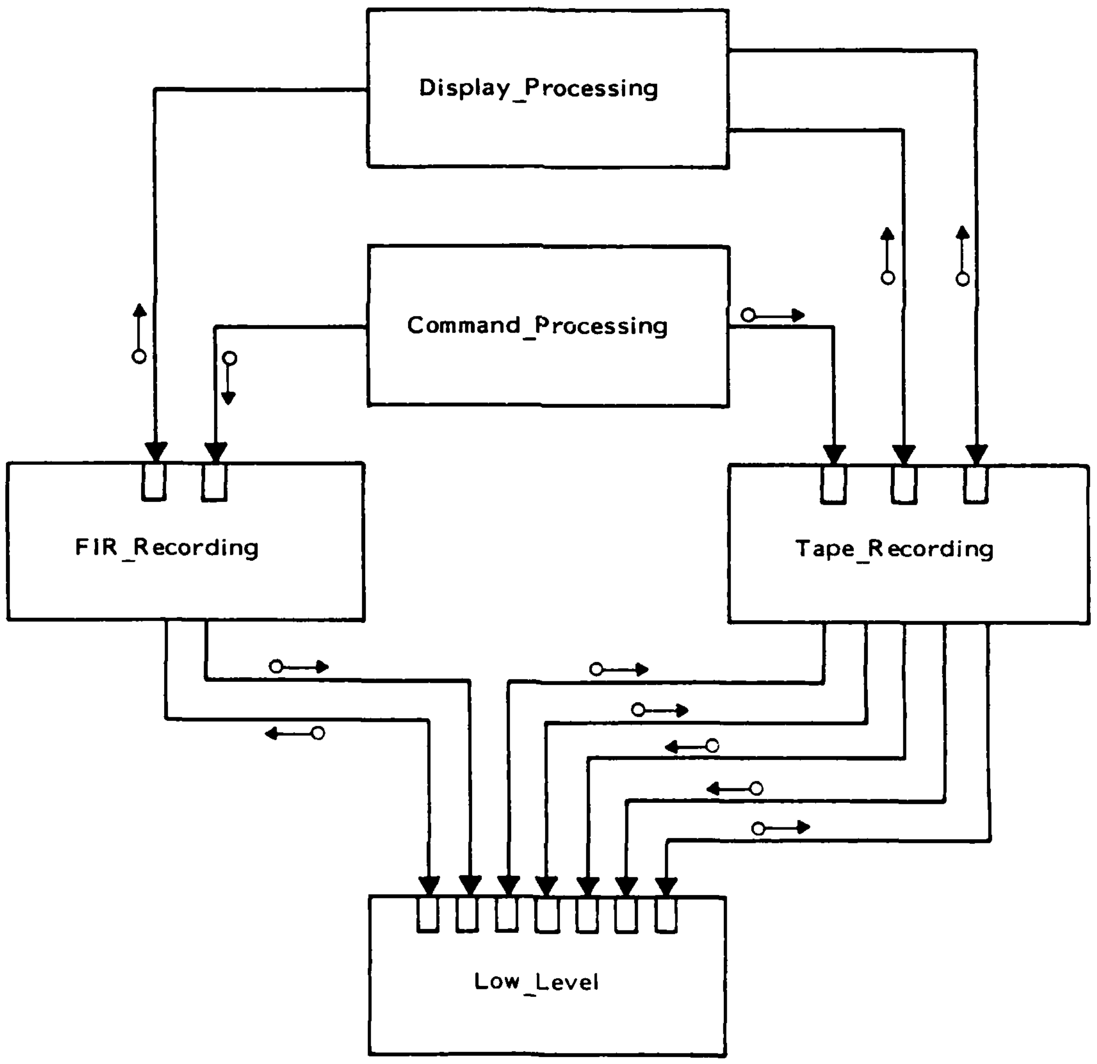
```
with Low_Level;
with Data_to_Cyclic;
with Data_to_Background;
with Data_Definitions;
use Data_Definitions;
with Data_to_Event;
use Data_to_Event;

package body Event_Driven_Package is    -- The routines in this package are run
                                        -- when the correct external events
                                        -- occur.
Local_Contact_Gate: Gate_Type;
Local_Event_Gate: Gate_Type;
Local_Extraction_Gate: Gate_Type;

Local_Targets: Target_Pair_Type;

  procedure Open_Gate;
  procedure Close_Gate;
  procedure Open_Target;
  procedure Close_Target;

  task body Batch_Handler is      -- This task does all the system maintenance
                                  -- that has to be performed every 1.5 degrees
                                  -- and associated processing.  This includes
                                  -- opening and closing gates and targets, and
                                  -- Extracting event and contact data.
  begin
    loop  -- forever
      accept Batch_Interrupt;
      if New_Background_Data then
        Local_Contact_Gate := Contact_Gate;
        Local_Event_Gate := Event_Gate;
        Local_Extraction_Gate := Extraction_Gate;
        New_Background_Data := False;
      end if;
      if New_Cyclic_Data then
        Local_Targets := Targets;
        New_Cyclic_Data := False;
      end if;
      Close_Gate;
      Open_Gate;
      Close_Target;
      Open_Target;
      if Local_Contact_Gate.Active and (not Low_Level.FIFO_Empty) then
        if Data_to_Cyclic.Contact_Switch = 1 then
         -- Get data i active input buffer
          Low_Level.FIFO_Get(Data_to_Cyclic.Contact_Buffers(2));
        else
          Low_Level.FIFO_Get(Data_to_Cyclic.Contact_Buffers(1));
      end if;
    end if;
  end loop;
end Batch_Handler;


procedure Open_Gate is separate;
  -- Open any gates in global file that are due to
  -- be opened by the next batch.
  -- if contact gate is opened, initialize contact buffer not pointed to
  -- by Contact_Switch unless data has not been emptied from it yet,
  -- in which case declare an error.

procedure Close_Gate is separate;
  -- Close any gates in global file that are due to
  -- be closed by the next batch.  If an event gate is closed
  -- read event file into event buffer not pointed to by Event_Switch
  -- unless there is unread data there, in which case declare an error.
  -- if contact gate is closed switch contact buffers if data has been read
  -- from other buffer, otherwise declare warning.

procedure Open_Target is separate;
  -- Open any targets in global file that are due to
  -- be opened by the next batch.

procedure Close_Target is separate;
  -- Close any targets in global file that are due to
  -- be closed by the next batch.


task body FIR_Handler is      -- This routine informs the rest of the system
                              -- that FIR data is ready to be extracted.
begin
  loop  -- forever
    accept FIR_Interrupt;
    Data_to_Cyclic.FIR_Ready := True;
  end loop;
end FIR_Handler;


task body Disk_End is         -- This informs various software that a DMA to
                              -- disk operation has completed.
begin
  loop  -- forever
    accept Disk_Interrupt;
    Low_Level.Disk_IO_Complete;
    Data_to_Cyclic.Disk_is_Done := True;
  end loop;
end Disk_End;
```

```
  function is_a_Command(Command: in Command_String) return Boolean;
  procedure Command_Error(Command: in Command_String);

  task body Keyboard_Handler is    -- This routine accepts characters until a
                                   -- complete command is formed, then sends the
                                   -- command to command processing.
    Input_Character : Character;
    Command         : Command_String;
    Length          : Integer := 0;
  begin
    loop  -- forever
      accept New_Character;
      Input_Character := Low_Level.Get_Character;
      if Length < 40 then
        Length := Length + 1;
        Command(Length) := Input_Character;
        if is_a_Command(Command) then
          if not(Data_to_Background.New_Command) then
            Data_to_Background.Command := Command;
            Data_to_Background.New_Command := True;
          else
            Command_Error(Command);
          end if;
          Command := (1..40 => " ");
          Length := 0;
        else
          Command_Error(Command);
        end if;
      end if;
    end loop;
  end Keyboard_Handler;

  function is_a_Command(Command: in Command_String) return Boolean is separate;
  procedure Command_Error(Command: in Command_String) is separate;


end Event_Driven_Package;
```

APPENDIX G

DATA DRIVEN DESIGN SOLUTION

The data driven design solution follows a structure as shown below:



DATA DRIVEN DESIGN SOLUTION
Data Definition

```
package Data_Types is

  type Data_Word_Type is array(1..16) of Boolean;
  pragma pack (Data_Word_Type);
  type Data_Holder_Type is array(Integer range <>) of Data_Word_Type;

  subtype Data_Array_Type is Data_Holder_Type(1..32767);
  type Data_Array_Pointer is access Data_Array_Type;
  subtype Control_Words_Type is Data_Holder_Type(1..5);

  subtype Contact_Buffer_Type is Data_Holder_Type(1..2**11);
  type Contact_Buffer_Pointer is access Contact_Buffer_Type;
  subtype Small_Contact_Type is Data_Holder_Type(1..42);

  subtype Event_Buffer_Type is Data_Holder_Type(1..10);
  type Event_Buffer_Pointer is access Event_Buffer_Type;

  type Tape_Mode_Type is (Events, Contacts, Both);

  type Slant_Ranges is array(1..2) of Float range 0.0..Float'Last;

  subtype Bearing_Type is Float range 0.0..360.0;

  type Bearings is array(1..2) of Bearing_Type;
----------------------------------------------------------

  type Gate_Type is
    record
      Active               : Boolean := False;
      On                   : Boolean := False;
      Range_Parameters     : Slant_Ranges;
      Bearing_Parameters : Bearings;
    end record;
  type Selection_Type is (Target, Clutter);
  type Target_Type is
    record
      Active                 : Boolean := False;
      On                     : Boolean := False;
      Selection              : Selection_Type;
      Range_Parameters       : Slant_Ranges;
      Range_Rate             : Float;
      Range_Acceleration     : Float;
      Bearing_Parameters     : Bearings;
      Bearing_Rate           : Float;
      Bearing_Acceleration : Float;
    end record;

  type Target_Pair_Type is array (1..2) of Target_Type;
end Data_Types;
```

```
with Data_Types;
use Data_Types;
package Low_Level is
  procedure FIR_In(Data_Array: out Data_Array_Pointer);
  procedure Disk_Out(Data_Array: in Data_Array_Pointer);
  procedure Contacts_In(Contact_Buffer: out Contact_Buffer_Pointer);
  procedure Events_In(Event_Buffer: out Event_Buffer_Pointer);
  procedure Contacts_Out(Contact_Buffer: in Contact_Buffer_Pointer);
  procedure Events_Out(Event_Buffer: in Event_Buffer_Pointer);
  function FIFO_Empty return Boolean;
  function Get_Character return Character;
end Low_Level;
```

```
with Low_Level;
with Data_Types;
use Data_Types;
package FIR_Recording is          -- This package handles the extraction
                                  -- recording of FIR data in this system.

  procedure Get_Control_Words(Words: out Control_Words_Type);
  pragma Inline(Get_Control_Words);
   -- This procedure returns the control words of the last FIR buffer for
   -- display purposes.
  procedure Turn_Disk_On;
  pragma Inline(Turn_Disk_On);
   -- This procedure enables disk recording as soon as data is ready.
  procedure Turn_Disk_Off;
  pragma Inline(Turn_Disk_Off);
   -- This procedure disables disk recording, and if it is re-enabled,
   -- recording will begin at the beginning of the disk.

end FIR_Recording;
```

DATA DRIVEN DESIGN SOLUTION
Package Specifications

```
with Low_Level;
with Data_Types;
use Data_Types;
package Tape_Recording is      -- This package comprises all the software
                               -- needed to extract the proper data and record
                               -- it to tape.

  procedure Get_Contact_Data(Contacts: out Small_Contact_Type);
  pragma Inline(Get_Contact_Data);
   -- This procedure returns a set of up to eight contacts for display purposes
  procedure Get_Event_Data(Events: out Event_Buffer_Type);
  pragma Inline(Get_Event_Data);
   -- This procedure returns a set of event data for display purposes.
  procedure Trigger_Contact_Fetch;
  pragma Inline(Trigger_Contact_Fetch);
   -- This triggers a contact fetch cycle, which will empty out any contact
   -- data from the FIFO.
  procedure End_Contact_Cycle;
  pragma Inline(End_Contact_Cycle);
   -- This procedure triggers a contact fetch cycle, which will empty out any
   -- contact data in the FIFO and swap contact buffers between tape recording
   -- and contact input.
  procedure Trigger_Event_Fetch;
  pragma Inline(Trigger_Event_Fetch);
   -- This triggers an event fetch in which event data will be read into the
   -- event input buffer and the buffers will be swapped between tape output
   -- and event input.
  procedure Turn_Tape_On(Tape_Recording_Mode: in Tape_Mode_Type);
  pragma Inline(Turn_Tape_On);
   -- This will enable tape recording.
  procedure Turn_Tape_Off;
  pragma Inline (Turn_Tape_Off);
   -- This disables tape recording, and if recording is re-enabled, it will
   -- start at the beginning of the tape.

end Tape_Recording;
```

```
package Command_Processing is  -- This package handles the characters coming
                               -- from the keyboard, recognizes commands and
                               -- distributes instructions to the rest of the
                               -- system.

  subtype Command_String is String(1..40);

  task Command_Processor;

  task Keyboard_Handler is
    entry New_Command(Next_Command: out Command_String);
    entry New_Character;
  --for New_Character use at 8#100204#;
  end Keyboard_Handler;

end Command_Processing;



package Display_Processing is   -- This package collects display data from the
                                -- rest of the system and displays the
                                -- appropriate set as determined by command
                                -- processing.
  procedure Update_Display(New_Display: in Boolean;
                           Display_Number: in Integer);
end Display_Processing;



with Data_Types;
use Data_Types;
package Radar_Insertion is      -- This package handles all the opening and
                                -- closing of gates and targets in the system.

  procedure New_Targets(Targets: in Target_Pair_Type);
  -- This procedure inputs new targets to be inserted into the Radar.
  pragma Inline (New_Targets);
  procedure New_Gates(Contact_Gate: in Gate_Type;
                      Event_Gate: in Gate_Type;
                      Extraction_Gate: in Gate_Type);
  -- This procedure inputs new Gates that are to be inserted into the Radar.
  pragma Inline (New_Gates);

end Radar_Insertion;
```

DATA DRIVEN DESIGN SOLUTION
Low Level I/O Package

```
package body Low_Level is
  procedure FIR_In(Data_Array: out Data_Array_Pointer) is separate;
  procedure Disk_Out(Data_Array: in Data_Array_Pointer) is separate;
  procedure Contacts_In(Contact_Buffer: out Contact_Buffer_Pointer) is separate;
  procedure Events_In(Event_Buffer: out Event_Buffer_Pointer) is separate;
  procedure Contacts_Out(Contact_Buffer: in Contact_Buffer_Pointer) is separate;
  procedure Events_Out(Event_Buffer: in Event_Buffer_Pointer) is separate;
  function FIFO_Empty return Boolean is separate;
  function Get_Character return Character is separate;
end Low_Level;
```

```
package body FIR_Recording is

  task FIR_Data_Ready_Handler is
                                    -- This task keeps track of the state of the
                                    -- FIR memory interface, enabling data
                                    -- transfers or interface tests when the
                                    -- conditions are right.
    entry FIR_Ready;
    entry Next_FIR;
    entry FIR_Transfer_Complete;
    entry Interface_Test_Request;
    entry Interface_Test_Complete;
  --for FIR_Ready use at 8#100300#;
  end FIR_Data_Ready_Handler;

  task Disk_Access_Controller is    -- This task monitors the state of the disk
                                    -- interface, allowing access when it is not
                                    -- already in use.
    entry Access_Granted;
    entry Access_Finished;
  end Disk_Access_Controller;

  task Display_Buffer is
                                    -- This task maintains a set of the most
                                    -- recent FIR control data.
    entry Data_Update(New_Control_Words: in Control_Words_Type);
    entry Data_Fetch(Current_Control_Words: out Control_Words_Type);
  end Display_Buffer;

  task Transfer_to_Disk is
                                    -- This Task Performs the actual transfer
                                    -- to the Disk if the disk is enabled.
    entry Disk_On;
    entry Disk_Off;
    entry Data_to_Disk(Data_Array: in Data_Array_Pointer);
    entry Transfer_Complete;
  --for Transfer_Complete use at 8#100301#;
  end Transfer_to_Disk;

  task type FIR_Extraction;

  Extraction_Tasks: array(1..4) of FIR_Extraction;

---------------------------------------------------------------

  procedure Get_Control_Words(Words: out Control_Words_Type) is
  begin
    Display_Buffer.Data_Fetch(Words);
  end Get_Control_Words;
  procedure Turn_Disk_On is
  begin
    Transfer_to_Disk.Disk_On;
  end Turn_Disk_On;

  procedure Turn_Disk_Off is
  begin
    Transfer_to_Disk.Disk_Off;
  end Turn_Disk_Off;

  task body FIR_Extraction is      -- A family of tasks each of which contains
                                   -- an FIR data buffer.  Each Task fills its
                                   -- data buffer when enabled and sends it to
                                   -- the disk when enabled.
    Data_Array    : Data_Array_Pointer;
    i             : Integer;
  begin
    Data_Array := new Data_Array_Type;
    loop -- forever
      FIR_Data_Ready_Handler.Next_FIR;
       -- Wait for data to be placed in FIR memory for extraction
      Low_Level.FIR_In(Data_Array);
       -- Get data in Data_Array
      FIR_Data_Ready_Handler.FIR_Transfer_Complete;
       -- Inform Controller that FIR transfer is done
      Display_Buffer.Data_Update(Data_Array(16380..16384));
       -- Send Control words to the display buffer
      Disk_Access_Controller.Access_Granted;
       -- Wait for disk access
      Transfer_to_Disk.Data_To_Disk(Data_Array);
       -- Send FIR data to disk
      Disk_Access_Controller.Access_Finished;
       -- Inform the disk Controller that the transfer to disk is finished.
    end loop;
  end FIR_Extraction;

  task body FIR_Data_Ready_Handler is    -- This task informs the system when
                                         -- FIR data is ready and the FIR
                                         -- interface is available for data
                                         -- retrieval.  It also coordinates FIR
                                         -- interface testing with normal data
                                         -- transfers.
    Data_is_Present         : Boolean := False;
    Interface_is_Available  : Boolean := True;
  begin
    loop  -- forever
      select
        accept FIR_Ready;
        Data_is_Present := True;
      or
```

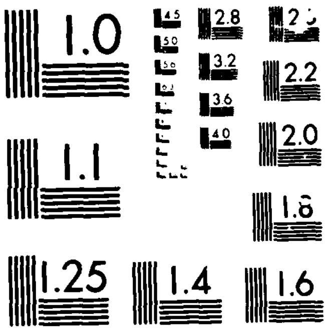
```
        when (Data_is_Present and Interface_is_Available) =>
        accept Next_FIR;
        Data_is_Present := False;
        Interface_is_Available := True;
      or
        accept FIR_Transfer_Complete;
        Interface_is_Available := True;
      or
        when Interface_is_Available =>
        accept Interface_Test_Request;
        Interface_is_Available := False;
      or
        accept Interface_Test_Complete;
        Interface_is_Available := True;
      end select;
    end loop;
  end FIR_Data_Ready_Handler;

  task body Disk_Access_Controller is       -- This task informs the system
                                            -- when the disk is available
                                            -- for data storage.
    Disk_is_Available : Boolean := True;
  begin
    loop  -- forever
      select
        when Disk_is_Available =>
        accept Access_Granted;
        Disk_is_Available := False;
      or
        accept Access_Finished;
        Disk_is_Available := True;
      end select;
    end loop;
  end Disk_Access_Controller;

  task body Display_Buffer is        -- This task maintains the control words
                                     -- from the most recent FIR buffer for
                                     -- Display Processing.
    Control_Words : Control_Words_Type;
  begin
    loop  --forever
      select
        accept Data_Update(New_Control_Words: in Control_Words_Type) do
          Control_Words := New_Control_Words;
        end Data_Update;
      or
        accept Data_Fetch(Current_Control_Words: out Control_Words_Type) do
          Current_Control_Words := Control_Words;
        end Data_Fetch;
      end select;
    end loop;
  end Display_Buffer;

  task body Transfer_to_Disk is    -- This task performs the transfer of data
                                   -- to the disk.
    Disk_is_On : Boolean := False;
  begin
    loop  -- forever
      select
        accept Disk_On;
        Disk_is_On := True;
      or
        accept Disk_Off;
        Disk_is_On := False;
      or
        accept Data_to_Disk(Data_Array: in Data_Array_Pointer) do
          if Disk_is_On then
            Low_Level.Disk_Out(Data_Array);
            accept Transfer_Complete;
            -- wait until transfer complete
          end if;
        end Data_to_Disk;
      end select;
    end loop;
  end Transfer_to_Disk;

end FIR_Recording;
```

```
package body Tape_Recording is

  task Contact_Data_Ready_Handler is  -- This task informs Contact Extraction
                                      -- tasks when it is time to fill their
                                      -- buffers.
    entry Next_Contact;
    entry Contact_Gate_Close;
  end Contact_Data_Ready_Handler;

  task Event_Data_Ready_Handler is    -- This task informs Event Extraction
                                      -- tasks when it is time to fill their
                                      -- buffers.
    entry Next_Event;
    entry Event_Gate_Close;
  end Event_Data_Ready_Handler;

  task Tape_Access_Controller is      -- This Task monitors the tape interface
                                      -- and allows data to flow to the tape
                                      -- output based on tape readiness and
                                      -- tape output mode.
    entry Tape_On;
    entry Tape_Off;
    entry Contact_Check_In;
    entry Event_Check_In;
    entry Contact_Access;
    entry Event_Access;
    entry Access_Finished;
    entry Tape_Trigger(Transfer_Type: out Tape_Mode_Type;
                       Tape_On: out Boolean);
    entry Set_Mode(Tape_Recording_Mode: in Tape_Mode_Type);
  end Tape_Access_Controller;

  task Display_Buffer is              -- This Task maintains a set of the most
                                      -- recent data for display.
    entry Contact_Update(Contacts: in Small_Contact_Type);
    entry Event_Update(Events: in Event_Buffer_Type);
    entry Contact_Fetch(Contacts: out Small_Contact_Type);
    entry Event_Fetch(Events: out Event_Buffer_Type);
  end Display_Buffer;

  task Transfer_to_Tape is            -- This task performs the transfer to
                                      -- tape if enabled.
    entry Contacts_to_Tape(Contacts: in Contact_Buffer_Pointer);
    entry Events_to_Tape(Events: in Event_Buffer_Pointer);
    entry Transfer_Complete;
  --for Transfer_Complete use at 8#100302#;
  end Transfer_to_Tape;

  task Extraction_Triggers is         -- This task triggers actual data
                                      -- extraction as needed.
    entry Contact_Time;
    entry Event_Time;
    entry Contact_Cycle;
    entry Contact_Go(Cycle_Continuing: out Boolean);
    entry Event_Go;
  end Extraction_Triggers;

  task type Contact_Extraction;

  Contact_Extraction_Tasks: array(1..2) of Contact_Extraction;

  task type Event_Extraction;

  Event_Extraction_Tasks: array(1..2) of Event_Extraction;

----------------------------------------------------------------

  procedure Get_Contact_Data(Contacts: out Small_Contact_Type) is
  begin
    Display_Buffer.Contact_Fetch(Contacts);
  end Get_Contact_Data;

  procedure Get_Event_Data(Events: out Event_Buffer_Type) is
  begin
    Display_Buffer.Event_Fetch(Events);
  end Get_Event_Data;

  procedure Trigger_Contact_Fetch is
  begin
    Extraction_Triggers.Contact_Time;
  end Trigger_Contact_Fetch;

  procedure End_Contact_Cycle is
  begin
    Extraction_Triggers.Contact_Cycle;
  end End_Contact_Cycle;

  procedure Trigger_Event_Fetch is
  begin
    Extraction_Triggers.Event_Time;
  end Trigger_Event_Fetch;

  procedure Turn_Tape_On(Tape_Recording_Mode: in Tape_Mode_Type) is
  begin
    Tape_Access_Controller.Set_Mode(Tape_Recording_Mode);
    Tape_Access_Controller.Tape_On;
  end Turn_Tape_On;

  procedure Turn_Tape_Off is
  begin
```

```
    Tape_Access_Controller.Tape_Off;
  end Turn_Tape_Off;


  task body Contact_Extraction is       -- This a family of tasks each one of
                                        -- which contains a contact buffer.
                                        -- Contacts are read into the buffer
                                        -- then sent to transfer to tape for
                                        -- final output.
    Contact_Buffer            : Contact_Buffer_Pointer;
    Contact_Extraction_Active : Boolean := False;
  begin
    Contact_Buffer := new Contact_Buffer_Type;
    loop -- forever
      Contact_Data_Ready_Handler.Next_Contact;
      Contact_Extraction_Active := True;
      While Contact_Extraction_Active = True loop
        Extraction_Triggers.Contact_Go(Contact_Extraction_Active);
        Low_Level.Contacts_In(Contact_Buffer);
      end loop;
      Contact_Data_Ready_Handler.Contact_Gate_Close;
      Tape_Access_Controller.Contact_Check_In;
      Tape_Access_Controller.Contact_Access;
      Transfer_to_Tape.Contacts_to_Tape(Contact_Buffer);
      Tape_Access_Controller.Access_Finished;
      Display_Buffer.Contact_Update(Contact_Buffer(1..42));
    end loop;
  end Contact_Extraction;

  task body Event_Extraction is         -- This a family of tasks each one of
                                        -- which contains an event buffer.
                                        -- Events are read into the buffer
                                        -- then sent to transfer to tape for
                                        -- final output.
    Event_Buffer : Event_Buffer_Pointer;
  begin
    Event_Buffer := new Event_Buffer_Type;
    loop -- forever
      Event_Data_Ready_Handler.Next_Event;
      Extraction_Triggers.Event_Go;
      Low_Level.Events_In(Event_Buffer);
      Event_Data_Ready_Handler.Event_Gate_Close;
      Tape_Access_Controller.Event_Check_In;
      Tape_Access_Controller.Event_Access;
      Transfer_to_Tape.Events_To_Tape(Event_Buffer);
      Tape_Access_Controller.Access_Finished;
      Display_Buffer.Event_Update(Event_Buffer.all);
    end loop;
  end Event_Extraction;
  task body Contact_Data_Ready_Handler is   -- This task tells Contact
                                            -- Extraction tasks when to pickup
                                            -- Data.
  begin
    loop  -- forever
      accept Next_Contact;
      accept Contact_Gate_Close;
    end loop;
  end Contact_Data_Ready_Handler;

  task body Event_Data_Ready_Handler is     -- This task tells Event
                                            -- Extraction tasks when to pickup
                                            -- Data.
  begin
    loop  -- forever
      accept Next_Event;
      accept Event_Gate_Close;
    end loop;
  end Event_Data_Ready_Handler;

  task body Extraction_Triggers is          -- This task receives timing
                                            -- triggers from outside, and
                                            -- determines when actual data
                                            -- fetch operations are to be
                                            -- performed.
  begin
    loop
      select
        accept Contact_Time;
        accept Contact_Go(Cycle_Continuing: out Boolean) do
          Cycle_Continuing := True;
        end Contact_Go;
      or
        accept Event_Time;
        accept Event_Go;
      or
        accept Contact_Cycle;
        accept Contact_Go(Cycle_Continuing: out Boolean) do
          Cycle_Continuing := False;
        end Contact_Go;
      end select;
    end loop;
  end Extraction_Triggers;

  task body Tape_Access_Controller is      -- This Task keeps track of the tape
                                           -- interface and allows data through
                                           -- to the tape output as dictated by
                                           -- Tape availability and Tape Mode.
    Tape_is_On        : Boolean := False;
    Tape_is_Available : Boolean := True;
```

AD-A169 687 DESIGNING REAL-TIME SYSTEMS IN ADA (TRADEMARK)(U) 3/3
SOFTECH INC WALTHAM MA 1986

UNCLASSIFIED F/G 9/2 NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.8
3.2
3.6
4.0
4.5
5.0

```
  Events_Checked    : Integer := 0;
  Contacts_Checked  : Integer := 0;
  End_Count         : Integer := 0;
  Tape_Mode         : Tape_Mode_Type;
begin
  loop  -- forever
    select
      accept Tape_On;
      Tape_is_On := True;
    or
      accept Tape_Off;
      Tape_is_On := False;
    or
      accept Set_Mode(Tape_Recording_Mode: in Tape_Mode_Type) do
        Tape_Mode := Tape_Recording_Mode;
      end Set_Mode;
    or
      accept Event_Check_In;
      Events_Checked := Events_Checked + 1;
    or
      accept Contact_Check_In;
      Contacts_Checked := Contacts_Checked + 1;
    or
      when (Tape_is_Available and
           (Tape_Mode = Events and Events_Checked > 0))  =>
        accept Event_Access;
        Tape_is_Available := False;
        End_Count := 1;
        Events_Checked := Events_Checked - 1;
        accept Tape_Trigger(Transfer_Type: out Tape_Mode_Type;
                            Tape_On: out Boolean) do
          Transfer_Type := Events;
          Tape_On := Tape_is_On;
        end Tape_Trigger;
    or
      when (Tape_is_Available and
           (Tape_Mode = Contacts and Contacts_Checked > 0))  =>
        accept Contact_Access;
        Tape_is_Available := False;
        End_Count := 1;
        Contacts_Checked := Contacts_Checked - 1;
        accept Tape_Trigger(Transfer_Type: out Tape_Mode_Type;
                            Tape_On: out Boolean) do
          Transfer_Type := Contacts;
          Tape_On := Tape_is_On;
        end Tape_Trigger;
    or
      when (Tape_is_Available and
           (Tape_Mode = Both and
           ( Events_Checked > 0 and Contacts_Checked > 0)))  =>
          accept Event_Access;
          accept Contact_Access;
          Tape_is_Available := False;
          End_Count := 2;
          Events_Checked := Events_Checked - 1;
          Contacts_Checked := Contacts_Checked - 1;
          accept Tape_Trigger(Transfer_Type: out Tape_Mode_Type;
                              Tape_On: out Boolean) do
            Transfer_Type := Both;
            Tape_On := Tape_is_On;
          end Tape_Trigger;
      or
        accept Access_Finished;
        End_Count := End_Count - 1;
        if End_Count < 1 then
          Tape_is_Available := True;
        end if;
      end select;
    end loop;
  end Tape_Access_Controller;

  task body Display_Buffer is        -- This Task keeps copies of the latest
                                     -- data for the display process.  Up to
                                     -- 6 contacts an a full event buffer are
                                     -- maintained.
    Contact_Data : Small_Contact_Type;
    Event_Data   : Event_Buffer_Type;
  begin
    loop  --forever
      select
        accept Contact_Update(Contacts: in Small_Contact_Type) do
          Contact_Data := Contacts;
        end Contact_Update;
      or
        accept Event_Update(Events: in Event_Buffer_Type) do
          Event_Data := Events;
        end Event_Update;
      or
        accept Contact_Fetch(Contacts: out Small_Contact_Type) do
          Contacts := Contact_Data;
        end Contact_Fetch;
      or
        accept Event_Fetch(Events: out Event_Buffer_Type) do
          Events := Event_Data;
        end Event_Fetch;
      end select;
    end loop;
  end Display_Buffer;

  task body Transfer_to_Tape is      -- This task does the actual storage to
```

```
                                          -- tape, if it is enabled.
      Tape_is_On      : Boolean := False;
      Transfer_Mode   : Tape_Mode_Type;
      Dummy_Contacts  : Contact_Buffer_Pointer := new Contact_Buffer_Type;
      Dummy_Events    : Event_Buffer_Pointer := new Event_Buffer_Type;
    begin
      loop  -- forever
        Tape_Access_Controller.Tape_Trigger(Transfer_Mode, Tape_is_On);
        case Transfer_Mode is
        when Events  =>
          accept Events_to_Tape(Events: in Event_Buffer_Pointer) do
            if Tape_is_On then
              Low_Level.Events_Out(Events);
              accept Transfer_Complete;
              Low_Level.Contacts_Out(Dummy_Contacts);
            end if;
          end Events_to_Tape;
        when Contacts  =>
          accept Contacts_to_Tape(Contacts: in Contact_Buffer_Pointer) do
            if Tape_is_On then
              Low_Level.Events_Out(Dummy_Events);
              accept Transfer_Complete;
              Low_Level.Contacts_Out(Contacts);
              accept Transfer_Complete;
            end if;
          end Contacts_to_Tape;
        when Both  =>
          accept Events_to_Tape(Events: in Event_Buffer_Pointer) do
            if Tape_is_On then
              Low_Level.Events_Out(Events);
              accept Transfer_Complete;
            end if;
          end Events_to_Tape;
          accept Contacts_to_Tape(Contacts: in Contact_Buffer_Pointer) do
            if Tape_is_On then
              Low_Level.Contacts_Out(Contacts);
              accept Transfer_Complete;
            end if;
          end Contacts_to_Tape;
        end case;
      end loop;
    end Transfer_to_Tape;

  end Tape_Recording;
```

```
with Low_Level;
with Tape_Recording;
with FIR_Recording;
with Radar_Insertion;
with Display_Processing;
with Data_Types;
use Data_Types;
package body Command_Processing is    -- This package receives commands from the
                                      -- keyboard, and interprets them for the
                                      -- system.

  task body Command_Processor is      -- This task processes the commands after
                                      -- they have been read from the keyboard
                                      -- by the keyboard handler task.
    Command                : Command_String;
    Update_Display         : Boolean := False;
    New_Display            : Boolean := False;
    Display_Number         : Integer;
    New_Target             : Boolean := False;
    Local_Targets          : Target_Pair_Type;
    New_Gate               : Boolean := False;
    Local_Contact_Gate     : Gate_Type;
    Local_Event_Gate       : Gate_Type;
    Local_Extraction_Gate  : Gate_Type;
    Start_Tape             : Boolean := False;
    Stop_Tape              : Boolean := False;
    Tape_Mode              : Tape_Mode_Type := Both;
    Start_Disk             : Boolean := False;
    Stop_Disk              : Boolean := False;
  begin
    Keyboard_Handler.New_Command(Command);
      -- process command, if it is manipulating gate or target parameters
      -- for future gate output these changes are recorded in the local gate
      -- and target files.  Only one gate or target may be in the process of
      -- modification at any one time.  When the command completes the
      -- modification of a gate the new gate flag will be set.  When the
      -- command completes the modification of a target the new target flag
      -- will be set.  Changes to the display are commanded through the use
      -- of the New_Display, Update_Display, and Display_Number parameters.
      -- The decision to start or stop either type of data recording can
      -- also be made.  This decision will be reflected Start_Tape, Stop_Tape
      -- or Start_Disk, Stop_Disk.  If tape is to be started, the appropriate
      -- Tape_Mode will also be indicated.
      if New_Target then
        Radar_Insertion.New_Targets(Local_Targets);
        New_Target := False;
      end if;
      if New_Gate then
        Radar_Insertion.New_Gates(Local_Contact_Gate, Local_Event_Gate,
                                  Local_Extraction_Gate);
        New_Gate := False;
      end if;
      if Update_Display then
        Display_Processing.Update_Display(New_Display, Display_Number);
        Update_Display := False;
      end if;
      if Start_Disk then
        FIR_Recording.Turn_Disk_On;
      end if;
      if Stop_Disk then
        FIR_Recording.Turn_Disk_Off;
      end if;
      if Start_Tape then
        Tape_Recording.Turn_Tape_On(Tape_Mode);
      end if;
      if Stop_Tape then
        Tape_Recording.Turn_Tape_Off;
    end if;
  end Command_Processor;

  function is_a_Command(Command: in Command_String) return Boolean;
  procedure Command_Error(Command: in Command_String);

  task body Keyboard_Handler is     -- This task reads characters from the
                                    -- keyboard one character at a time until
                                    -- a full command is formed.
    Input_Character : Character;
    Command         : Command_String;
    Length          : Integer := 0;
    Command_Present : Boolean := False;
  begin
    loop  -- forever
      select
        when not(Command_Present) =>
          accept New_Character;
          Input_Character := Low_Level.Get_Character;
          if Length < 40 then
            Length := Length + 1;
            Command(Length) := Input_Character;
            if is_a_Command(Command) then
              Command_Present := True;
            end if;
          else
            Command_Error(Command);
          end if;
      or
        when Command_Present =>
          accept New_Command(Next_Command: out Command_String) do
            Next_Command := Command;
          end New_Command;
          Command_Present := False;
          Command := (1..40 =>' ');
          Length := 0;
      end select;
    end loop;
  end Keyboard_Handler;

  function is_a_Command(Command: in Command_String) return Boolean is separate;

  procedure Command_Error(Command: in Command_String) is separate;

end Command_Processing;
```

```
with FIR_Recording;
with Tape_Recording;
With Data_Types;
use Data_Types;
package body Display_Processing is  -- This package maintains the system
                                    -- display as directed by command
                                    -- processing.

  Local_Display_Number : Integer;
  Local_Control_Words  : Control_Words_Type;
  Local_Contact_Data   : Small_Contact_Type;
  Local_Event_Data     : Event_Buffer_Type;

  procedure Update_Display(New_Display: in Boolean;
                           Display_Number: in Integer) is
  begin
    if New_Display then
      Local_Display_Number := Display_Number;
    end if;
    FIR_Recording.Get_Control_Words(Local_Control_Words);
    Tape_Recording.Get_Contact_Data(Local_Contact_Data);
    Tape_Recording.Get_Event_Data(Local_Event_Data);
     -- update display information
  end Update_Display;

end Display_Processing;
```

DATA DRIVEN DESIGN SOLUTION
Radar Insertion Package

```
with FIR_Recording;
with Tape_Recording;
package body Radar_Insertion is     -- This package maintains the gates and
                                    -- targets in the system as directed by
                                    -- command processing.

  Task Gate_Tender is
    entry Input_Targets(Targets: in Target_Pair_Type);
    entry Input_Gates(Contact_Gate: in Gate_Type;
                      Event_Gate: in Gate_Type;
                      Extraction_Gate: in Gate_Type);
    entry Batch_Interrupt;
  --for Batch_Interrupt use at 8#100201#;
  end Gate_Tender;

  procedure Open_Gate;
  procedure Close_Gate;
  procedure Open_Target;
  procedure Close_Target;

  task body Gate_Tender is
    Local_Targets          : Target_Pair_Type;
    Local_Contact_Gate     : Gate_Type;
    Local_Event_Gate       : Gate_Type;
    Local_Extraction_Gate  : Gate_Type;
    Contact_Closed         : Boolean := False;
    Event_Closed           : Boolean := False;
  begin
    loop  -- forever
      select
        accept Batch_Interrupt;
        Close_Gate;
        Open_Gate;
        Close_Target;
        Open_Target;
        if Local_Contact_Gate.Active then
          Tape_Recording.Trigger_Contact_Fetch;
        end if;
        if Contact_Closed then
          Tape_Recording.End_Contact_Cycle;
          Contact_Closed := False;
        end if;
        if Event_Closed then
          Tape_Recording.Trigger_Event_Fetch;
          Event_Closed := False;
        end if;
      or
        accept Input_Targets(Targets: in Target_Pair_Type) do
          Local_Targets := Targets;
        end Input_Targets;
      or
        accept Input_Gates(Contact_Gate: in Gate_Type;
                           Event_Gate: in Gate_Type;
                           Extraction_Gate: in Gate_Type) do
          Local_Contact_Gate := Contact_Gate;
          Local_Event_Gate := Event_Gate;
          Local_Extraction_Gate := Extraction_Gate;
        end Input_Gates;
      end select;
    end loop;
  end Gate_Tender;

  procedure New_Targets(Targets: in Target_Pair_Type) is
  begin
    Gate_Tender.Input_Targets(Targets);
  end New_Targets;

  procedure New_Gates(Contact_Gate: in Gate_Type;
                      Event_Gate: in Gate_Type;
                      Extraction_Gate: in Gate_Type) is
  begin
    Gate_Tender.Input_Gates(Contact_Gate, Event_Gate, Extraction_Gate);
  end New_Gates;

  procedure Open_Gate is separate;
    -- Open any gates in global file that are due to
    -- be opened by the next batch.

  procedure Close_Gate is separate;
    -- Close any gates in global file that are due to
    -- be closed by the next batch.  If an event gate is closed
    -- Event_Closed flag is set.  If contact gate is closed Contact_Closed flag
    -- is set.

  procedure Open_Target is separate;
    -- Open any targets in global file that are due to
    -- be opened by the next batch.

  procedure Close_Target is separate;
    -- Close any targets in global file that are due to
    -- be closed by the next batch.

end Radar_Insertion;
```

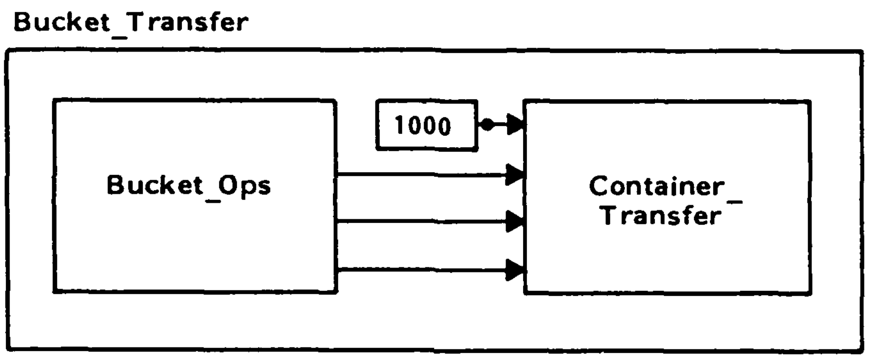
APPENDIX H

CODING ISSUES

This appendix discusses some of the specific coding issues encountered during the production of the software listed in chapter 9. 1 - Interface Dummy Read. In the sample Low Level package listed in Appendix A it is necessary to perform a dummy read before reading any FIR location in order to get the interface into the proper state. In order to ensure that the compiler will not optimize out this read it was necessary to declare this as a shared variable. This is one of the few cases where even though the data is never used, the read cannot be optimized out. 2 - Shared Data - The data passing between threads of control in the cyclic executive example is done by means of shared data. This is done to limit the impact of the running of one thread of control on the scheduling of another. When it is time for a task to run it can never be blocked due to an attempted rendezvous; data may not be ready, but the control can still be shifted and any operations not requiring the missing data can still be performed. Shared data has many drawbacks. It causes problems in large designs and during the maintenance phase of the software life cycle. Each shared data item must be accessed in a very strict fashion. Any deviation from the correct access method is likely to cause serious data coordination problems. Typically problems arise when insufficient documentation of the data passing strategy is provided, or the strategy is changed, and not every one is told, or the strategy fails to account for one or more important cases. These reasons are why data passing is usually accomplished though more structured means such as rendezvous. Shared data was used in this example to illustrate a common practice in cyclic executive systems.

APPENDIX I

DETAILED DESIGN OF THE BUFFER

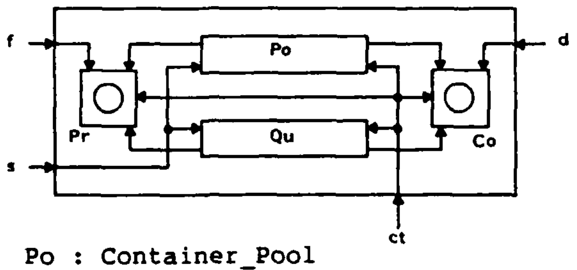


```
with Container_Transfer, Bucket_Ops;
package Bucket_Transfer is
        new Container_Transfer(
            1000,
            Bucket_Ops.Bucket_Type;
            Bucket_Ops.Fill_Bucket;
            Bucket_Ops.Drain_Bucket);
```

DETAILED DESIGN OF THE BUFFER
Bucket_Ops

```
package Bucket_Ops is
    type Bucket_Type is private;
    procedure Fill_Bucket (B : in out Bucket_Type);
    procedure Drain_Bucket(B : in out Bucket_Type);
private
    type Bucket_Record;
    type Bucket_Type is access Bucket_Record;
end Bucket_Ops;
```

T : Container_Transfer



s	:	Buffer_Size
ct	:	Container_Type
f	:	Fill_Container
d	:	Drain_Container

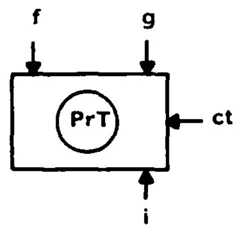
Po	:	Container_Pool
Qu	:	Container_Queue
Pr	:	Container_Producer
Co	:	Container_Consumer

```
generic
    Buffer_Size : Positive;
    type Container_Type is private;
    with procedure Fill_Container  (C : in out Container_Type);
    with procedure Drain_Container (C : in out Container_Type);
package Container_Transfer is
end Container_Transfer;
```

DETAILED DESIGN OF THE BUFFER
Container_Transfer

```
with
    Container_Pool, Container_Queue,
    Container_Producer, Container_Consumer;
package body Container_Transfer is
    package Container_Pool_1 is
        new Container_Pool(Buffer_Size, Container_Type);
    package Container_Queue_1 is
        new Container_Queue(Buffer_Size, Container_Type);
    package Container_Producer_1 is
        new Container_Producer(
                                Container_Type,
                                Container_Pool_1.Get_Container,
                                Fill_Container,
                                Container_Queue_1.Insert_Container);
    package Container_Consumer_1 is
        new Container_Consumer(
                                Container_Type,
                                Container_Queue_1.Get_Container,
                                Drain_Container,
                                Container_Pool_1.Return_Container);
end Container_Transfer;
```

Pr : Container_Producer



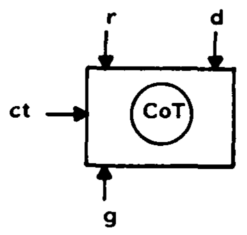
```
ct : Container_Type;
f  : Fill_Container;
g  : Get_Container;
i  : Insert_Container;
```

PrT : Producer_Task;

```
generic
    type Container_Type is private;
    with procedure Get_Container    (C : out    Container_Type);
    with procedure Fill_Container   (C : in out Container_Type);
    with procedure Insert_Container(C : in     Container_Type);
package Container_Producer is
    task Producer_Task is
        entry Data_Ready;
    end Producer_Task;
end Container_Producer;

package body Container_Producer is
    task body Producer_Task is
        C : Container_Type;
    begin
        loop
            accept Data_Ready;
            Get_Container(C);
            Fill_Container(C);
            Insert_Container(C);
        end loop;
    end Producer_Task;
end Container_Producer;
```

DETAILED DESIGN OF THE BUFFER
Container_Consumer

Co : Container_Consumer;

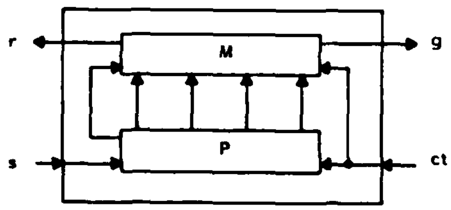


```
ct  : Container_Type;
r   : Return_Container;
g   : Get_Container;
d   : Drain_Container;
```

CoT : Consumer_Task;

```
generic
    type Container_Type is private;
    with procedure Get_Container    (C : out    Container_Type);
    with procedure Drain_Container  (C : in out Container_Type);
    with procedure Return_Container (C : in out Container_Type);
package Container_Consumer is
    task Consumer_Task;
end Container_Consumer;

package body Container_Consumer is
    task body Consumer_Task is
        C : Container_Type;
    begin
        loop
            Get_Container(C);
            Drain_Container(C);
            Return_Container(C);
        end loop;
    end Consumer_Task;
end Container_Consumer;
```

```
Po : Container_Pool;
```

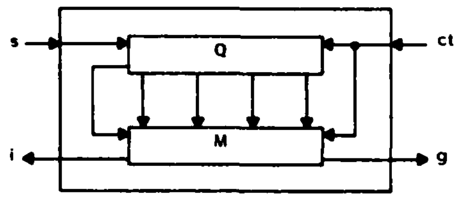


```
s  : Buffer_Size;
ct : Container_Type;
g  : Get_Container
r  : Return_Container;
```

```
M : Monitor;
P : Container_Pool_Ops;


generic
    Buffer_Size : Positive;
    type Container_Type is private;
package Container_Pool is
    procedure Get_Container   (C : out    Container_Type);
    procedure Return_Container(C : in out Container_Type);
end Container_Pool;


with Monitor, Container_Pool_Ops;
package body Container_Pool is
    package Container_Pool_Ops_1 is
        new Container_Pool_Ops(Buffer_Size, Container_Type);
    package Monitor_1 is
        new Monitor(
            Container_Type,
            Container_Pool_Ops_1.Container_Pool_Type,
            Container_Pool_Ops_1.Is_Empty,
            Container_Pool_Ops_1.Is_Full,
            Container_Pool_Ops_1.Get_From_Pool,
            Container_Pool_Ops_1.Put_Into_Pool);

    procedure Get_Container    (C : out    Container_Type) is
    begin
        Monitor_1.Get_Container(C);
    end Get_Container;

    procedure Return_Container (C : in out Container_Type) is
    begin
        Monitor_1.Put_Container(C);
    end Return_Container;
end Container_Pool;
```

DETAILED DESIGN OF THE BUFFER
Container_Queue

Qu : Container_Queue;

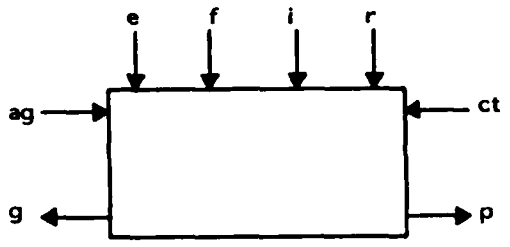


s	:	Buffer_Size;
ct	:	Container_Type;
i	:	Insert_Container;
g	:	Get_Container;

M : Monitor;
Q : Container_Queue_Ops;

```
generic
    Buffer_Size : Positive;
    type Container_Type is private;
package Container_Queue is
    procedure Insert_Container (C : in out  Container_Type);
    procedure Get_Container    (C : out     Container_Type);
end Container_Queue;


package body Container_Queue is
    package Container_Queue_Ops_1 is
        new Container_Queue_Ops(Buffer_Size, Container_Type);
    package Monitor_1 is
        new Monitor(
            Container_Type,
            Container_Queue_Ops_1.Container_Queue_Type,
            Container_Queue_Ops_1.Is_Empty,
            Container_Queue_Ops_1.Is_Full,
            Container_Queue_Ops_1.Get_From_Queue,
            Container_Queue_Ops_1.Put_Into_Queue);

    procedure Insert_Container (C : in out  Container_Type) is
    begin
        Monitor_1.Put_Container(C)
    end Insert_Container;

    procedure Get_Container    (C : out     Container_Type) is
    begin
        Monitor_1.Get_Container(C);
    end Get_Container;
end Container_Queue;
```

M : Monitor;

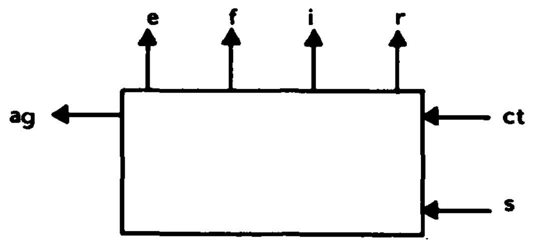


ct	:	Container_Type;
ag	:	Aggregate_Type;
e	:	Is_Empty;
f	:	Is_Full;
i	:	Insert_Into_Aggregate;
r	:	Remove_From_Aggregate;
p	:	Put_Container;
g	:	Get_Container;

```
generic
    type Container_Type is private;
    type Aggregate_Type is private;
    with function Is_Empty (A : Aggregate_Type) return Boolean;
    with function Is_Full  (A : Aggregate_Type) return Boolean;
    with procedure Insert_Into_Aggregate(
            C : in  Container_Type; A : in out Aggregate_Type);
    with procedure Remove_From_Aggregate(
            C : out Container_Type; A : in out Aggregate_Type);
package Monitor is
    procedure Put_Container(C : in out Container_Type);
    procedure Get_Container(C : in     Container_Type);
end Monitor;
```

DETAILED DESIGN OF THE BUFFER
Container_Pool_Ops

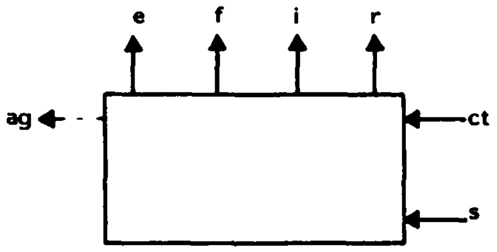
P : Container_Pool_Ops;



s	:	Buffer_Size;
ct	:	Container_Type;
ag	:	Container_Pool_Type;
e	:	Is_Empty;
f	:	Is_Full;
i	:	Put_Into_Pool;
r	:	Get_From_Pool;

```
generic
    Buffer_Size : Positive;
    type Container_Type is private;
package Container_Pool_Ops is
    type Container_Pool_Type is private;
    function  Is_Empty (P : Container_Pool_Type) return Boolean;
    function  Is_Full  (P : Container_Pool_Type) return Boolean;
    procedure Get_From_Pool(
        C : out Container_Type; P : in out Container_Pool_Type);
    procedure Put_Into_Pool(
        C : in  Container_Type; P : in out Container_Pool_Type);
private
    type Container_Pool_Record;
    type Container_Pool_Type is access Container_Pool_Record;
end Container_Pool_Ops;
```

Q : Container_Queue_Ops;



s	: Buffer_Size;
ct	: Container_Type;
ag	: Container_Queue_Type;
e	: Is_Empty;
f	: Is_Full;
i	: Put_Into_Queue;
r	: Get_From_Queue;

```
generic
    Buffer_Size : Positive;
    type Container_Type is private;
package Container_Queue_Ops is
    type Container_Queue_Type is private;
    function  Is_Empty (Q : Container_Queue_Type) return Boolean;
    function  Is_Full  (Q : Container_Queue_Type) return Boolean;
    procedure Get_From_Queue(
        C : out Container_Type; Q : in out Container_Queue_Type);
    procedure Put_Into_Queue(
        C : in  Container_Type; Q : in out Container_Queue_Type);
private
    type Container_Queue_Record;
    type Container_Queue_Type is access Container_Queue_Record;
end Container_Queue_Ops;
```

END
DTIC
8-86